VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ
FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

ÚSTAV TELEKOMUNIKACÍ
DEPARTMENT OF TELECOMMUNICATIONS

NÁVRH A IMPLEMENTACE ROZHRANÍ PRO ZPRACOVÁNÍ RÁMCŮ XPON

DESIGN AND IMPLEMENTATION OF XPON FRAME PROCESSING INTERFACE

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. Zdeněk Vais**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. Martin Holík**
SUPERVISOR

BRNO 2020

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Diplomová práce

magisterský navazující studijní obor **Telekomunikační a informační technika**

Ústav telekomunikací

Student: Bc. Zdeněk Vais ***ID:*** 117829
Ročník: 2 ***Akademický rok:*** 2019/20

NÁZEV TÉMATU:

Návrh a implementace rozhraní pro zpracování rámců xPON

POKYNY PRO VYPRACOVÁNÍ:

Cílem diplomové práce je návrh a implementace řešení, které umožní ukládání a manipulaci s rámci z GPON (Gigabyte Passive Optical Network) sítě. V teoretické části diplomové práce se seznamte a popište sítě GPON a NG-PON a jejich rozdíly. V praktické části diplomové práce využijte databázový systém MongoDB, který umožňuje práci s daty v JSON (JavaScript Object Notation) formátu. Navrhněte a naprogramujte sadu skriptů v jazyce Python pro manipulaci s rámci v datovém úložišti, zejména pro paralelizaci zápisových operací.

DOPORUČENÁ LITERATURA:

[1] PLUGGE, Eelco, Peter MEMBREY a David HOWS. MongoDB Basics. 1. Apress, 2014. ISBN 978-1-4-4-20895-3.

[2] CHODOROW, Kristina. MongoDB: the definitive guide. Second edition. Beijing: O'Reilly, [2013]. ISBN 978--449-34468-9.

Termín zadání: 3.2.2020 ***Termín odevzdání:*** 1.6.2020

Vedoucí práce: Ing. Martin Holík

prof. Ing. Jiří Mišurec, CSc.
předseda oborové rady

ABSTRAKT

Tato práce je zaměřena na systém pro uchování GPON komunikace. V práci je teoreticky rozebrána problematika související s optickými sítěmi GPON a NG-PON, NoSQL databází a databázovým systémem MongoDB. Praktickým výstupem práce je návrh databáze a zdrojové kódy v jazycích Python a C# pro práci s touto databází. Práce je zakončena výkonnostním testováním, které prokazuje, že navržená databáze a zdrojové kódy jsou použitelné v reálném provozu.

KLÍČOVÁ SLOVA

Pasivní optická síť, GPON, databáze, NoSQL, MongoDB, JSON, Python, C#

ABSTRACT

This thesis focuses on system for persistence of GPON communication. Theoretical part deals with problems of GPON and NG-PON optical networks, NoSQL database systems and MongoDB database. Practical part contains design of database schema for MongoDB database and source code in programming languages Python and C# for working with this database. The thesis is finalized by performance testing, proving that the database design and source code implementation is capable of handling real world traffic.

KEYWORDS

Passive optical network, GPON, database system, NoSQL, MongoDB, JSON, Python, C#

VAIS, Zdeněk. *Návrh a implementace rozhraní pro zpracování rámců xPON*. Brno, 2020, 108 s. Diplomová práce. Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav telekomunikací. Vedoucí práce: Ing. Martin Holík

PROHLÁŠENÍ

Prohlašuji, že svou diplomovou práci na téma „Návrh a implementace rozhraní pro zpracování rámců xPON“ jsem vypracoval samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor uvedené diplomové práce dále prohlašuji, že v souvislosti s vytvořením této diplomové práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a/nebo majetkových a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů, včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

Brno ....

podpis autora

PODĚKOVÁNÍ

Rád bych poděkoval vedoucímu diplomové práce panu Ing. Martinu Holíkovi za odborné vedení, konzultace, trpělivost a podnětné návrhy k práci.

Obsah

Seznam obrázků

Seznam tabulek

Seznam výpisů

Úvod

Tato práce se zabývá uchováním síťového provozu na GPON sítích a nepřímo navazuje na vývoj FPGA karty sloužící k zachycení a analýze této komunikace, který byl proveden na Ústavu telekomunikací Fakulty elektrotechniky VUT v Brně.

Cílem je navržení databázového úložiště s použitím databázového systému MongoDB. A dále vytvořit sadu zdrojových kódů v jazyce Python a C#, které by umožňovaly práci s touto databází.

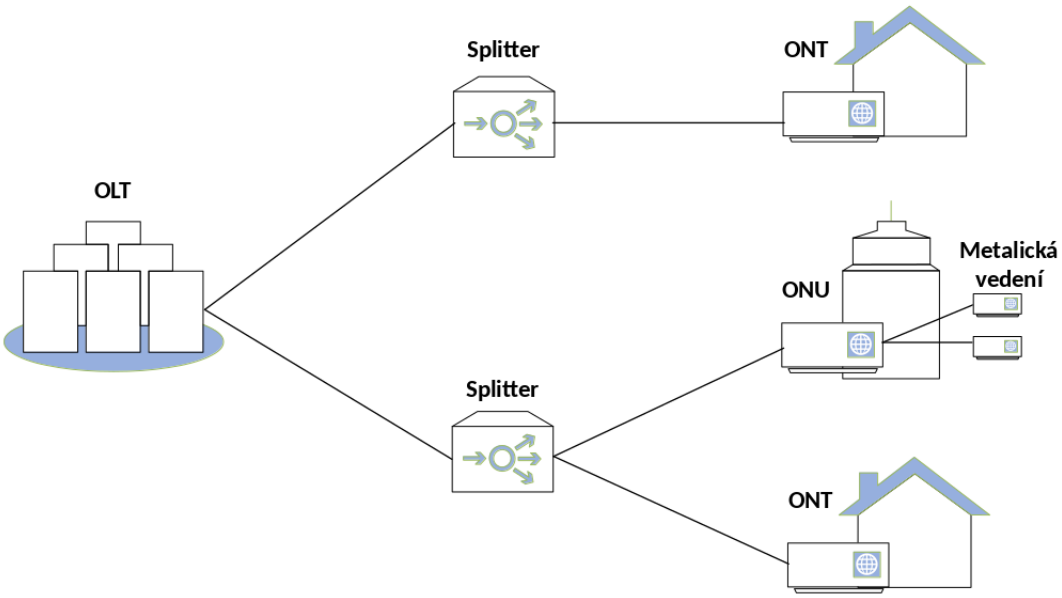
Vzhledem k vysoké rychlosti optických sítí se jako klíčový jeví požadavek na rychlost zápisu. Tomuto požadavku se podřizuje i návrh schématu databáze. K dosažení nejvyšší možné rychlosti využívají zdrojové kódy hromadné ukládání, asynchronní operace a paralelizaci. Na konci práce je provedeno výkonnostní měření, výběr nejvhodnějšího použití vytvořených zdrojových kódů a diskuze o použitelnosti v reálném provozu.

1 Pasivní optické sítě

Optické sítě se dělí na aktivní (AON - Active Optical Network) a pasivní (PON - Passive Optical Network). Aktivní sítě vyžadují, aby některé ze síťových prvků byly napájené, a využívají se především u transportních sítí. Naopak prvky pasivní sítě nevyžadují žádné napájení a díky své nízké pořizovací ceně nacházejí uplatnění především jako tzv. řešení poslední míle k připojení koncových uživatelů. Časté jsou také sítěmi kombinujícími metalická a optická vedení, které vznikají postupným nahrazováním metalických vedení. [1]

1.1 Topologie sítě

Pasivní sítě jsou typu Point-to-multipoint. Na straně poskytovatele se vyskytuje jedno zařízení, ke kterému je přes síťové prvky připojeno více zařízení koncových uživatelů.



Obr. 1.1: Topologie PON sítí.

Samotná síť sestává z následujících prvků, viz obr. 1.1:

- **ONT** (Optical Network Terminal) je optickým modemem na straně koncového uživatele. Slouží k překladu optického signálu pro jednotlivá zařízení jako např.

počítače, televize, VoIP apod. Tato zařízení mají často integrovaný WiFi modem [21] [7] [8].

- **ONU** (Optical Network Unit) je stejně jako ONT zařízení, které slouží k ukončení optické sítě. Stará se o překlad optického signálu na elektrický. Oproti ONT však zajišťuje připojení více uživatelů [7] [8].
- **Splitter** rozděluje signál z jednoho optického vlákna na více [7] [8].
- **OLT** (Optical Line Terminate) je zařízení na straně poskytovatele internetu, kontroluje veškerou komunikaci GPON sítě. Zajišťuje fungování sítě na 1., 2. a 3. síťové vrstvě a propojení s vnější sítí. Dále zajišťuje správu koncových zařízení ONT a ONU [21] [7] [8] .

.

2 Standard GPON a XG-PON

Jedním z prvních standardů pro komunikaci v PON sítích byl APON (ATM PON), po něm následoval BPON (Broadband PON), který zajišťoval až 622 Mb/s v sestupném směru (downstream). V současnosti zažívá rozmach standard ITU-T G.984 známý jako GPON (Gigabit Passive Optical Network) zajišťující až 2,5 Gb/s. Nástupcem GPON je standard ITU-T G.987 známý též jako XG-PON nebo 10G-PON, který by měl umožnit až 10 Gb/s v sestupném směru a 2,5 Gb/s ve vzestupném směru (upstream) [1] [20].

Jak standard GPON, tak XG-PON, využívá v sestupném směru dělení na rámce metodou TDM (Time Division Multiplex). Velikost rámců tedy není přesně daná, ale je ohraničená časovými úseky o velikosti 125 μs. U vzestupného směru zůstává zachován koncept časového dělení na úseky 125 μs, avšak je doplněn pravidelnými časovými tiky/značkami, které slouží k synchronizaci. Aby byla umožněna obousměrná komunikace, používají se pro vzestupný a sestupný směr různé vlnové délky. U GPON je pro sestupný směr je použit signál o vlnové délce 1490 nm a pro vzestupný 1310 nm a u XG-PON 1577 nm v sestupném směru a 1270 nm v směru vzestupném.

Oba standardy využívají zapouzdřování protokolů do vrstev, z nichž každá má svůj účel [1]:

- Servisní adaptační vrstvu,
- GEM/XGEM rámce,
- GTC/XGTC rámce a
- rámce fyzické vrstvy.

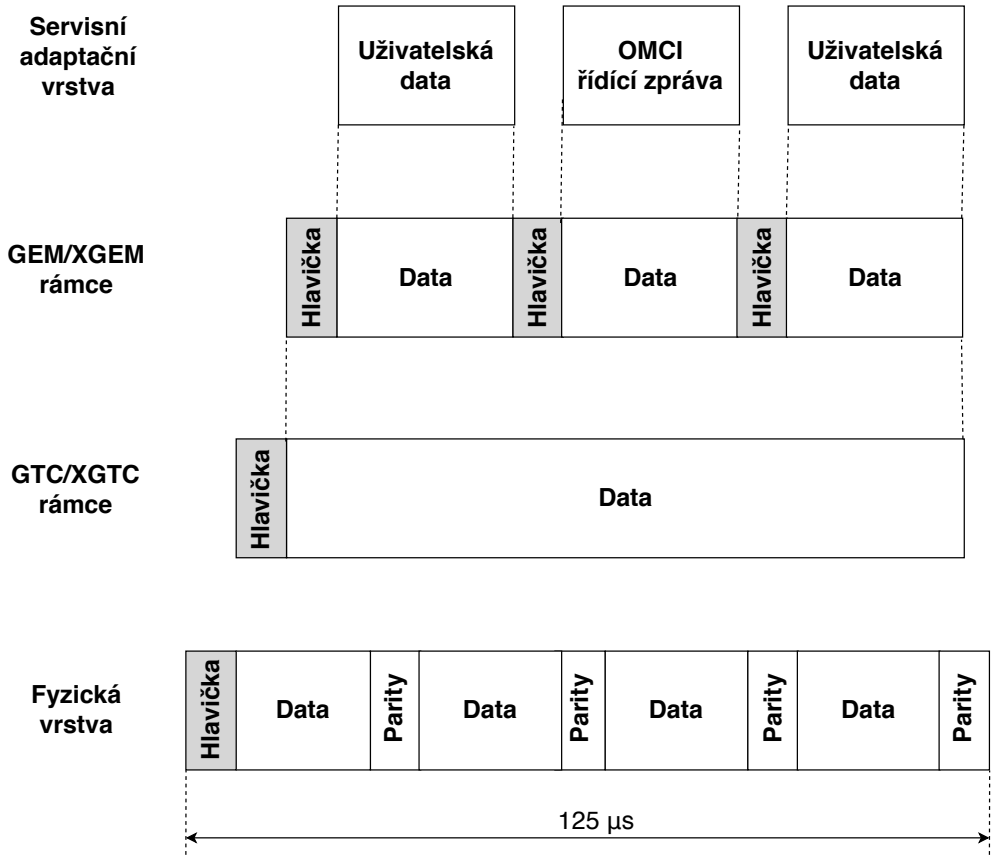
Jednotlivé vrstvy jsou znázorněny na obr. 2.1.

2.1 Servisní adaptační vrstva

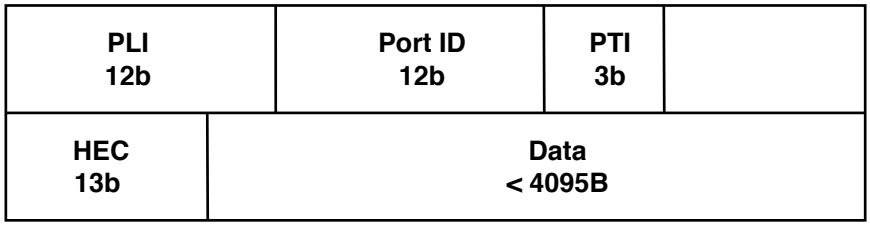
Servisní vrstva (Service Adaptation Layer) přenáší rámce zvané Service data unit (SDU), které obsahují buďto data uživatelů, nebo řídící zprávy OMCI (ONU Management and Control Interface). Jedním z hlavních účelů této vrsty je poskytnout nezávislost na vrstvách pod ní, a proto má stejný format pro GPON i XG-PON.

2.2 Rámce GEM a XGEM

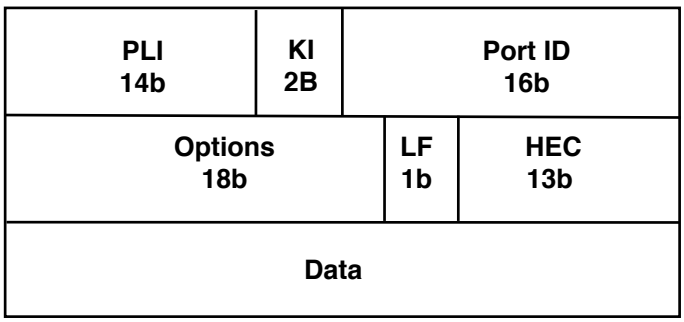
Na této vrstvě se struktura rámců pro GPON a XG-PON liší. Struktura GEM rámce je znázorněná na obr. 2.2 a rámce XGEM na obr. 2.3:



Obr. 2.1: Zapouzdřování vrstev u PON



Obr. 2.2: Struktura GEM rámce



Obr. 2.3: Struktura XGEM rámce

2.2.1 Společná pole

Ač se velikost některých polí liší, jejich význam zůstává zachován pro oba standardy. Příklady takových polí jsou:

- **PLI** (Payload Length Identificator) určuje délku dat v bytech v poli Payload. U GPONu je maximální hodnota tohoto pole 4095 a u XG-PONu 16383.
- **Port ID** identifikuje typ provozu a slouží k předání pomocných informací o multiplexování.
- **HEC** (Header Error Correction) obsahuje kontrolní součet sloužící k ověření správného dekódování hlavičky rámce.

2.2.2 GPON pole

Jediným GPON specifickým polem je:

- **PTI** (Payload Type Identificator), které slouží k identifikaci typu dat přenášených v poli Payload a jejich fragmentaci. Význam jednotlivých hodnot je uveden v tabulce 2.1:

2.2.3 XG-PON pole

U XG-PON došlo k odstranění pole PTI, které bylo nahrazeno polem KI a LF. Význam jednotlivých polí je následující:

- **KI** neboli Key index určuje jakým klíčem je zašifrované pole Payload. 00 specifikuje, že pole zašifrované není, 01 je první klíč, 10 je druhý klíč a hodnota 11 je prozatím nevyužitá.
- **Options** prozatím nevyužité pole, slouží k zarovnání hlavičky na 64 bitů, což umožňuje snažší překlad na protokol GPON.

Tab. 2.1: Význam hodnot pole PTI

000	Uživatelská data, fragment není poslední
001	Uživatelská data, poslední fragment
010	Rezervováno a nepoužito
011	Rezervováno a nepoužito
100	Řídící zpráva OAM, fragment není poslední
101	Řídící zpráva OAM, poslední fragment
110	Rezervováno a nepoužito
111	Rezervováno a nepoužito

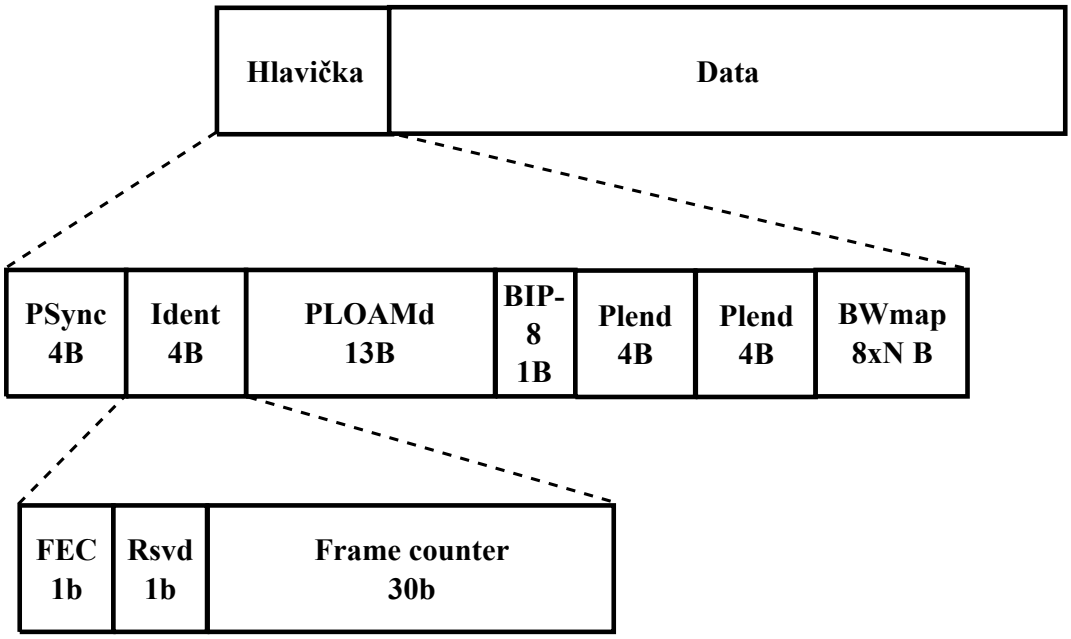
- **LF** (Last fragment) slouží k fragmentaci SDU. Pokud je LF roven 1, tak se jedná o poslední fragment a nebo SDU fragmentováno není vůbec.

2.3 Rámce GTC a XGTC

GTC/XGTC vrstva zapozdřuje GEM/XGEM rámce a OLT ji využívá k přiřazení rozsahu pro vzestupný směr jednotlivým ONU. Stejně jako u GEM a XGEM rámců, i zde je možné najít podobnosti mezi oběma protokoly. Rámec protokolu GTC je znázorněn na obr. 2.4 a XGTC na obr. 2.6:

2.3.1 GTC

- **PSync** je sekvence bitů, kterou ONU využívá k synchronizaci fyzické vrstvy.
- **Ident** sestavá z tří dílčích částí:
  - *FEC* (Forward Error Correction) je jeden bit, který definuje, zda-li je použito protichybové kódování. Pokud je hodnota 1, je použita.
  - *Rsvd* (Reserved) je nepoužívaný bit.
  - *Frame counter* je třicet bitů sloužících k počítání rámců. Po dosažení maximální hodnoty se opět začíná od nuly.
- **PLOAMd** (Physical Layer Operation and Maintenance downstream) obsahuje pole řídící zprávy mezi OLT a ONU/ONT, z nichž každá je dlouhá 13 bajtů. Tyto zprávy slouží zejména k aktivaci a deaktivaci ONU a ONT, správě jejich stavů a přiřazení ONU-ID.
- **BIP-8** (Bit Interleaved Parity 8) je soubor paritních bitů vypočítaný z předchozích tří polí.
- **Plend** sestavá ze tří úseků:
  - *Blen* určuje počet prvků v Bandwith mapě.



Obr. 2.4: Rámec GTC

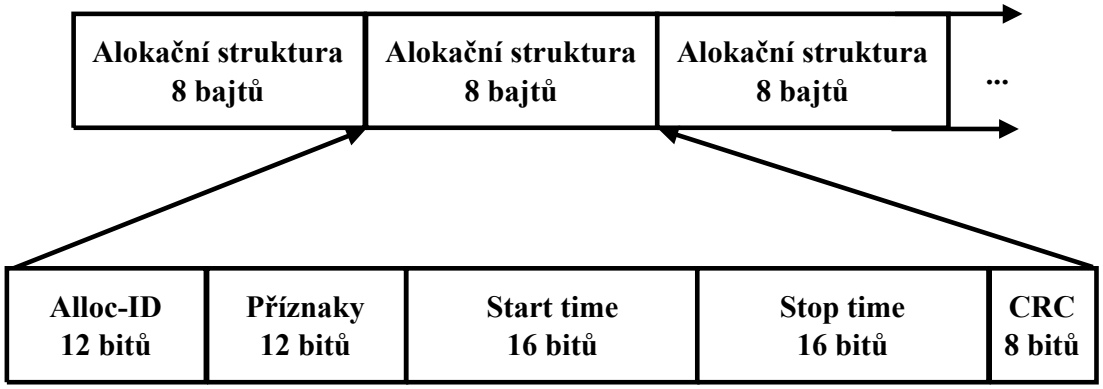
  - *Alen* je dnes již nepoužívané pole, které je vždy nula.
  - *CRC* je kontrolní součet.

  Toto pole se záměrně v rámci opakuje.
- **BWmap** neboli Bandwith mapa slouží k přidělování kapacity pro provoz v vzestupném (upstream) směru. Více v následující sekci.

GTC bandwith mapa

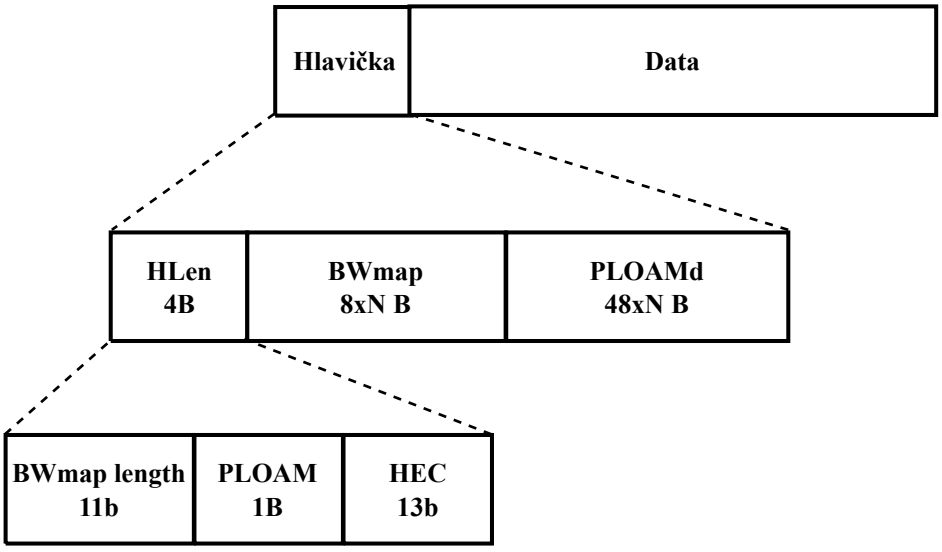
Toto pole slouží k přiřazení práv pro zasílání dat ve vzestupném směru pro jednotlivá ONU. Bandwith mapa sestává z tzv. alokačních struktura, z nichž každá má délku 8 bajtů, viz obr. 2.5.

- **Alloc-ID** určuje, komu jsou přiřazena práva na vzestupný přenos. Hodnoty menší než 254 odpovídají ONU-ID. Hodnota 254 slouží pro neznámá zařízení.
- **Flags** je příznakové pole určující parametry odchozího provozu.
- **Start time** určuje, kde začínají data pro přenos dané ONU. Měří se v bajtech od začátku vzestupného rámce.
- **Stop time** určuje, kde končí data pro přenos. Měří se v bajtech od začátku vzestupného rámce.
- **CRC** (Cyclic Redundancy Check) je kontrolní součet alokační struktury.



Obr. 2.5: Struktura pole BWmap u GTC

2.3.2 XGTC



Obr. 2.6: Rámec XGTC

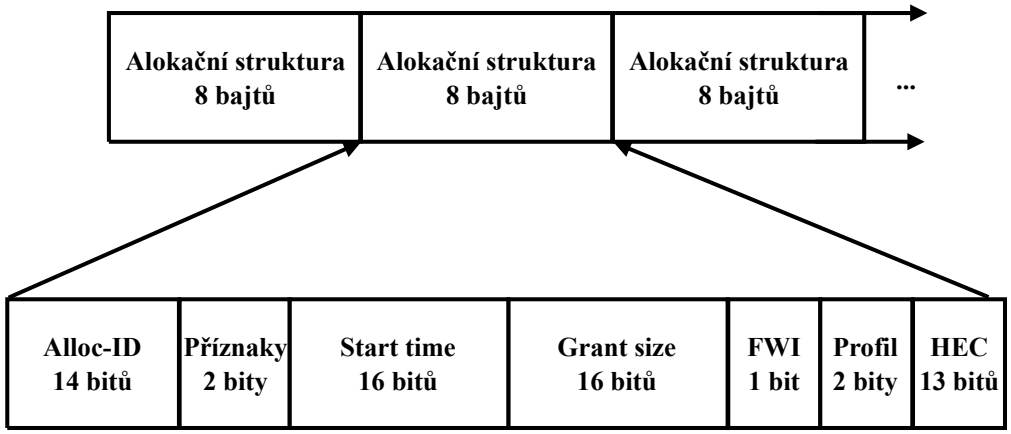
XGTC hlavička sestává z následujících polí, viz obr. 2.6:

- **HLend** je složené pole nesoucí informaci o délce dalších položek v hlavičce. Sestává z tří položek:
  - *BWmap length* které určuje počet alokačních struktur v poli BWmap (obdoba BWmap z GTC).
  - *PLOAM count* určující počet PLOAM zpráv.

  - HEC (Hybrid Error Correction) sloužící k ochraně celého pole HLend před chybami.
- **BWmap** stejně jako u GTC slouží k přiřazení práv pro vzestupný proud jednotlivým ONU. Struktura BWmap se však liší.
- **PLOAMd** má obdobný účel jako u GPON. Liší se však strukturou a zpráva má délku 48 bajtů.

XGTC bandwith mapa

XGTC bandwith mapa se od GTC poněkud liší. Její strukturu můžeme vidět na obr. 2.7.



Obr. 2.7: Struktura pole BWmapu XGTC

- **Alloc-ID** slouží k identifikaci ONU.
- **Flags** je příznakové pole pro parametry odchozího provozu.
- **Start time** určuje, kde začínají data pro přenos dané ONU ve vzestupném směru. Počítá se v počtu slov (4 bajty).
- **Grant size** určuje počet slov (4 bajty) vyhrazených pro data dané ONU ve vzestupném směru.
- **FWI** (Forced Wakup Indicator) se používá k probuzení zařízení, která podporují low-power mód.
- **Profile index** specifikuje jaký typ hlavičky bude použit pro komunikaci ve vzestupném směru. OLT definuje jeden nebo více profilů a předá je ONU. ONU bude používat jeden z těchto profilů ve svých odpovědích na odpovědích na vzestupnou komunikaci.
- **HEC** slouží k ochraně jedné alokační struktury.

3 NoSQL databáze

Ještě nedávno platilo, že pod pojmem "databáze" si většina odborníků představila relační centralizovanou databázi složenou z tabulek provázaných vazbami (relacemi). Relační databáze, také označované jako SQL (Structured Query Language) podle dotazovacího jazyka, který se pro ně používá, dominovaly informačním systémům dlouhá léta. Bylo to dáno zejména tím, že většina systémů vyžadovala zajištění konzistence data, podporu transakcí a normalizaci dat kvůli zmenšení místa na disku. Mezi systémy s těmito požadavky patřily a stále patří např. finanční a obchodní aplikace, ERP systémy apod. V posledních deseti letech však vzniká celá řada systémů a aplikací, pro něž zajištění konzistence a normalizace dat není prioritou, potřebují však zvládat zpracování extrémního množství dat. Mezi takové aplikace patří například systémy pro sběr a analýzu dat IoT zařízení, nástroje pro monitoring sítí nebo firemní infrastruktury atd. Relační databáze díky své povaze takové požadavky nemohou uspokojit buďto vůbec a nebo pouze za vysokou cenu hardwaru a licencí. Databázovým systémům, které nenásledují relační paradigma, se říká nerelační nebo NoSQL databáze [2][4].

3.1 Společné vlastnosti

Ač NoSQL databází existuje velmi velké množství lišící se datovým modelem, formátem ukládaných dat i funkcionalitou, je možné mezi nimi najít některé společné vlastnosti.

Jednou z motivací pro vznik a rozvoj NoSQL databází je flexibilita uložených dat. Samozřejmě i u tradičních SQL databází je možné změnit schéma za běhu, taková operace je však prováděna zdlouhavou a komplikovanou migrací již existujících dat, která nemusí uspět. Flexibilita schématu se u různých NoSQL systémů liší, avšak je zpravidla podstatně větší než u databází relačních. Jednotnost schématu a schopnost zpracovat uložená data se u NoSQL databází standardně zajišťuje aplikace, která ji využívá [2].

Dále NoSQL databáze díky menší nebo žádné podpoře konzistence a transakčnosti mají větší rychlost zápisu a snadnější horizontální škálovatelnost.

3.2 Typy NoSQL databází

Aby bylo možné se bavit i o rozdílných vlastnostech jednotlivých NoSQL databází, je potřeba nejdříve vymezit některé jejich základní typy. Nejčastěji bývají NoSQL databáze rozděleny na základě jejich datového modelu.

3.2.1 Databáze typu klíč-hodnota

Databáze klíč-hodnota (key-value), jak již jejich název napovídá, rozdělují datový model na dvě základní entity. Jednou je klíč, který zpravidla slouží k rychlému vyhledávání a případně i k určení úložiště dat u škálovatelných řešení. Hodnotou může být často cokoliv. Mezi tyto databáze patří například Redis nebo Couchbase [2, 10, 11].

3.2.2 Sloupcové databáze

Sloupcové databáze (Column store) si můžeme představit jako sled tabulek, avšak oproti relačním databázím nemají pevně daný počet sloupců, a je proto možné za běhu přidat libovolný sloupec bez nutnosti jeho přidání do existujících řádků. Velmi často se u těchto databází můžeme setkat s tím, že první sloupec je povinný a slouží jako jedinečný identifikátor, proto je možné tyto databáze zařadit i do kategorie klíč-hodnota. Typickými zástupci sloupcových databází jsou například Amazon DynamoDB, Cassandra nebo HBase [2, 12, 13, 14].

3.2.3 Objektové databáze

Objektové databáze ukládají objekty, které můžeme znát z Objektově orientovaného programování, a umožňují v nich vyhledávat na základě jejich vlastností. Tento přístup usnadňuje komunikaci s objektově naprogramovanou aplikací, neboť není nutná změna paradigmatu během práce s databází. Mezi objektové databáze patří např. ObjectivityDB nebo Zope DB [15, 16].

3.2.4 Dokumentové databáze

Tyto databáze ukládají různé druhy strukturovaných dokumentů, u nichž se předpokládá, že mají samopopisný charakter. Typickým příkladem těchto formátů je například XML nebo JSON (JavaScript Object Notation). Dokumentové databáze umožňují přístup k uloženým dokumentům a vyhledávání v jejich obsahu. Typickými zástupci jsou například MongoDB nebo Elasticsearch [2, 9].

3.2.5 Grafové databáze

Grafové databáze se od všech předchozích podstatně liší. Tento typ databáze je určen pro problémy, kdy je vhodné data ukládat jako grafy a také tak s nimi pracovat. Umožňují například vyhledávat sousedy nebo hledat cesty v grafu. Poslední dobou se z grafových databází prosazuje zejména Neo4J [2, 17].

4 MongoDB

MongoDB je dokumentová NoSQL databáze poprvé vydaná v roce 2009, která k ukládání dat používá formát BSON (Binary JSON), což je binární varianta populárního formátu JSON (JavaScript Object Notation). Jedná se o open source projekt s licencí GNU-Affero General Public License vyvíjený především společností MongoDB, Inc., která k němu nabízí placené služby. V současnosti MongoDB podporuje většinu hlavních operačních systémů včetně Windows, Linuxu, Macintosh a Solaris [19].

Koncepcí této databáze je poskytnout rychlé a škálovatelné úložiště s vysokou dostupností, avšak nabízí i některé funkce známé spíše ze světa relačních databází jako jsou například transakce či replikace dat [9, 19].

4.1 Komponenty

Samotná databáze sestává z několika základních komponent:

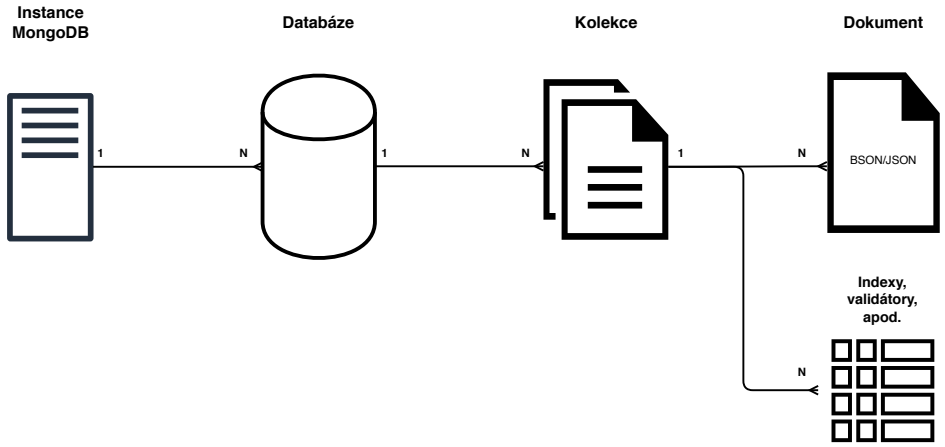
- **Core server** je proces databáze, který se stará o zpracování příkazů a poskytuje vlastní binární protokol pro jejich zasílání a zpětnou komunikaci. Na Windows se jedná o proces *mongodb.exe* a na Linuxu *mongod* [9].
- **Mongo shell** je primárním nástrojem pro práci s databází. Využívá programovací jazyk JavaScript a je z něj možné provádět všechny operace nad databází [9].
- **Konektory** pro různé programovací jazyky umožňující práci nejen z JavaScriptového shellu. MongoDB poskytuje ovladače pro většinu hlavních programovacích jazyků včetně C, C++, Pythonu, Javě, PHP, Scale atd. Ne vždy je však možné provádět všechny operace jako z Mongo shellu [9].
- **Nástroje příkazové řádky**, které slouží například pro zálohu a obnovu databáze (*mongodump* a *mongorestore*), import a export dat (*mongoimport* a *mongoimport*), ladění aplikací (*mongostat*) apod. [9].

4.2 Datový model

Jedna instance databázového serveru může obsahovat více databází, přičemž databáze obsahuje skupiny dokumentů, které jsou označovány jako kolekce. Kolekci si lze představit jako složku, která obsahuje soubory a na niž je možné navázat další databázové objekty jako jsou například indexy, validátory apod. Kolekce by měly obsahovat dokumenty ve stejném nebo podobném formátu, případně mohou sloužit

k vytvoření logických skupin dokumentů. Struktura jednotlivých dokumentů v kolekci není bez předchozí definice nijak kontrolována a každý dokument ji může mít jinou [9].

Každý z těchto objektů musí mít jedinečný název v rámci nadřazeného objektu. Kolekce tedy musí mít jedinečný název v rámci databáze, ve které se nacházejí, avšak ne v rámci celé MongoDB instance.



Obr. 4.1: Struktura MongoDB

4.2.1 Formát JSON a BSON

Z uživatelského hlediska databáze pracuje s formátem JSON. Data jsou čtena i vkládána v tomto formátu. Jedná se o úsporný pro člověka čitelný formát sestávající z dvojic klíč-hodnota, objektů a polí. Ukázka JSON formátu je vidět na příkladě 4.1 [22].

Výpis 4.1: Příklad JSON formátu

```
{
  "sampleObject":{
  "booleanProperty": true,
  "integerProperty": 42,
  "stringProperty": "abc",
  "stringArray": ["Text1", "Text2"],
  "objectArray":[
    {"id": "Identifier1"},
    {"id": "Identifier2"},
    {"id": "Identifier3"},
  ]
  }
}
```

Interně si MongoDB převádí data do binární obdoby fromátu JSON známé jako BSON (Binary JSON), která nezabírá tolik místa. K překladu z formátu JSON do BSON dochází ještě před odesláním dotazů do databáze [9].

4.2.2 Datové typy

Datové typy podporované databází vychází z formátu BSON, avšak ne ve všech verzích MongoDB musí být všechny BSON datové typy podoporovány. Typy podporované MongoDB ve verzi 4.2 jsou uvedeny v tabulce 4.1 [19].

Tab. 4.1: BSON datové typy

Typ	**Alias**	**Popis**
Celé číslo 32-bit	"int"	-
Celé číslo 64-bitů	"long"	-
Double	"double"	Desetinné číslo, 64 bitů
Decimal	"decimal"	Desetinné číslo, 128 bitů
String	"string"	Textový řetězec v UTF-8
Object	"object"	Zanořený objekt
Binární data	"binData"	Binární data v jakémkoliv formátu
Datum a čas	"date"	UNIXový čas, 64 bitů
Regulerní výraz	"regex"	-
Javascript	"javascriptWithScope"	-
ObjectId	"ObejctId"	-

ObjectId

Speciálním datovým typem je *ObjectId*, které se používá jako jedinečný identifikátor dokumentů v databázi. ObjectId je dlouhé 12 bajtů a je generováno ze tří částí:

- **Časová známka** (timestamp) je čtyřbajtová hodnota reprezentující počet sekund od Unixové epochy,
- *náhodná hodnota* o délce 5 bajtů a
- *čítač* o velikosti 3 bajty, který je inkrementován při každém vytvoření nového ObjectId [19].

4.3 Práce s databází

Práce s databází bude ilustrována v Mongo shellu. Mongo shell se spouští pomocí aplikace *mongo*, která se nachází ve složce *bin* v instalačním adresáři MongoDb.

4.3.1 Práce s databázovými objekty

Je důležité si uvědomit, že API MongoDB i Mongo shellu vychází z javascriptové objektové reprezentace, proto na většině databázových entit ať už to jsou databáze, kolekce nebo indexy jsou metody pro práci s nimi, jak je tomu zvykem u objektově orientovaného programování [9].

Příkaz show

K zobrazení objektů v databázi se používá příkaz *show*. Můžeme ho použít k výpisu již existujících databází s parametrem *databases* a nebo k vylistování kolekcí v aktuální databázi s parametrem *collections* [19].

Výpis 4.2: Výpis existujících databází a kolekcí v aktuální databázi

```
> show databases
> show collections
```

Příkaz use

Příkaz *use* slouží k přepnutí nebo vytvoření databáze. Takovéto chování je pro MongoDB standardní. MongoDB objekty vytváří zpravidla až v okamžiku, kdy jsou poprvé použity. Pokud bude parametrem příkazu *use* existující databáze, přepne se kontext na vybranou databázi, jestliže databáze neexistuje, bude vytvořena a opět dojde ke změně kontextu na tuto databázi. Po použití tohoto příkazu, bude možné využívat objekty dané databáze [19].

Výpis 4.3: Přepne nebo vytvoří databázi

```
> use database_name
```

Pokud nepoužijeme příkaz *use*, bude aktuální databáze první ze seznamu databází vypsaného příkazem *show*.

Proměnná db

K vypsání a práci s aktuální databází slouží proměnná *db*. Po zadání této proměnné do příkazové řádky bude vypsán název aktuální databáze. Proměnná *db* se zároveň používá k přístupu k objektům, které jsou na databázi vázané [19].

4.3.2 Práce s kolekcí

Stejný princip, který se uplaťnuje pro vytvoření databáze, platí i pro kolekce - kolekce je vytvořena až v okamžiku jejího použití. Jakýkoliv příkaz pro práci s daty z kapitoly 4.4 ji tedy vytvoří.

K přístupu ke kolekci se používá následující syntaxe:

Výpis 4.4: Syntaxe pro přístup k kolekci v databázi

```
> db.<nazev kolekce>
```

4.4 Práce s daty

V této kapitole bude popsáno, jak provádět základní operace pro čtení a manipulaci s daty/dokumenty (CRUD - Create, Read, Update, Delete) a jejich vyhledávání. Všechny tyto operace jsou reprezentovány metodami nad kolekcemi.

4.4.1 Vkládání nových dat

K vložení nových dat má kolekce tři metody:

- **insertOne()**, která vloží právě jeden záznam do kolekce a vrací jedinečný identifikátor nově vytvořeného záznamu.
- **insertMany()**, která slouží k vkládání více záznamů najednou a vrací identifikátory nově vzniklých záznamů
- **insert()** lze použít pro oba předchozí případy, avšak vrací pouze potvrzení, zda-li operace proběhla úspěšně.

Například k vložení jednoho záznamu do kolekce *courses* může sloužit následující příklad 4.5:

Výpis 4.5: Vložení jednoho záznamu do kolekce.

```
db.collection.insertOne(
  {
    course: "Programming",
    credits: 5,
    teacher: "John␣Doe"
  }
)
```

Pokud vložení nového záznamu proběhne úspěšně, bude vrácen velmi podobný dokument tomu z příkladu 4.6:

Výpis 4.6: Návratová hodnota po úspěšném vložení jednoho dokumentu.

```
{
  "acknowledged" : true,
  "insertedId" : ObjectId("5dfc2598e46b03e1f28a78cd")
}
```

Uložení více záznamů je znázorněno na příkladu 4.8. Za povšimnutí zde stojí, že vkládané záznamy nemají stejné schéma. To ničemu nevadí, jak již bylo řečeno flexibilita schématu je jedním ze stavebních kamenů NoSQL databází.

Výpis 4.7: Vložení více záznamů.

```
db.collection.insertMany(
  {
    course: "Programming",
    credits: 5,
    teacher: "John␣Doe"
  },
  {
    course: "Database␣systems",
    credits: 3,
    teachers: ["John␣Doe", "John␣Smith"]
  }
)
```

Pole _id

Ke každému nově vytvořenému záznamu je po vložení do databáze přiřazeno pole *_id* typu *ObjectId* (viz kapitola 4.2.2). Toto pole slouží jako jedinečný identifikátor a je zároveň indexováno.

4.4.2 Čtení a vyhledávání dat

K čtení a vyhledávání dat slouží funkce *find()*. Tuto funkci lze volat bez parametrů, v tom případě vrátí všechny záznamy v dané kolekci.

Výpis 4.8: Načtení všech záznamů z kolekce.

```
db.collection.find()
```

Metoda *find* však má i dva další nepovinné parametry:

- **filtrační parametry**, které slouží k omezení množiny vrácených dat a
- **projekce** dat sloužící k výběru dat z dokumentu. Pokud tento parametr není zvolen, jsou dokumenty vráceny tak, jak jsou uloženy v databázi.

Jako filtrační parametr se používá JSON objekt, který specifikuje podmínky na hodnoty v hledaném dokumentu. Pokud mají hledané dokumenty obsahovat přesně dané hodnoty, stačí jejich hodnoty zadat do filtračního objektu. Filtrace dat na základě jedinečného identifikátoru dokumentu _id je k vidění v příkladu 4.10

Výpis 4.9: Filtrace dat na základě jedinečného identifikátoru.

```
db.collection.find(
  {
    _id: ObjectId('4ecc05e55dd98a436ddcc47c')
  })
```

K filtraci dat z příkladu 4.8 je možné provést následujícím způsobem:

Výpis 4.10: Filtrace dat na základě jedinečného identifikátoru.

```
db.collection.find(
{
  course: "Programming"
})
```

Tab. 4.2: Srovnávací operátory MongoDB

Zápis	**Popis**
$eq	Rovná se
$ne	Nerovná se
$gt	Větší než
$gte	Větší nebo rovno
$lt	Menší než
$lte	Menší nebo rovno
$in	Ověřuje, jestli je prvek v poli
$nin	Ověřuje, jestli prvek není v poli

Tab. 4.3: Logické operátory MongoDB

Zápis	**Popis**
$and	Logický součin
$not	Negace
$or	Logický součet
$nor	Negace logického součinu

JSON operátory

Pro komplikovanější dotazy do databáze je již nutné použít JSON operátory, které databáze poskytuje. Každý operátor začíná symbolem $. Operátorů je poměrně velké množství a standardně se rozdělují do následujících kategorií:

- **Srovnávací** kontrolující rovnost, nerovnost, velikost dané hodnoty a zda-li je obsažena v poli.
- **Logické**, které můžeme znát z Booleovské algebry.
- **Bitové**, které pracují s bity.
- **Vyhodnocovací**, které nad hodnoty z dokumentu provádí určitou operaci. Sem patří především textové vyhledávání a regulerní výrazy.
- **Pro práci s poli**.
- **Vyhodnocující strukturu** dokumentu. Sem patří například operátory ověřující jestli dokument má určité pole.
- **Geolokační** pracující s polohou.
- **Změnu dat**.

Kompletní přehled operátorů je uveden v [19]. V tabulce 4.2 jsou uvedeny srovnávací operátory a v tabulce 4.3 logické.

Filtrace dat z příkladu 4.8 k získání kurzů s počtem kreditů větším než čtyři je vidět na příkladě 4.11.

Výpis 4.11: Filtrace s použitím operátorů.

```
db.collection.find(
  {
    credits: { $gt: 4 }
  })
```

Logické operátory fungují nad polem filtračních objektů. Příklad 4.12 ukazuje filtraci dat pro kredity, které jsou v rozmezí od 3 (včetně) po 5 (včetně).

Výpis 4.12: Filtrace s použitím více operátoru.

```
db.collection.find(
{
  $and:[
    { credits: { $gte: 3 } },
    { credits: { $lte: 5 } }
  ]
})
```

4.4.3 Změna dat

Ke změně dat slouží metoda *Update*, která má několik parametrů, z nichž první tři jsou nejdůležitější:

- **Filtrační parametry** jsou povinné a specifikují, která data se mají změnit a mají stejný formát jako v kapitole 4.4.2.
- **Definice změny** je povinný parametr, který specifikuje, co se má v nalezených dokumentech změnit.
- **Upsert** je nepovinný booleavský parametr. Pokud je *true* a nejsou nalezeny žádné dokumenty, které by měly být změněny, vloží do databáze nový dokument.

Kompletní seznam parametrů metody *Update* je uveden v její dokumentaci [19].

Změna dokumentu může být provedena několika způsoby. Pokud druhý parametr neobsahuje žádný změnový operátor, jsou nalezené záznamy nahrazeny obsahem druhého parametru. Nahrazení je demonstrováno v příkladu 4.13, kde první parametr filtruje kurzy s názvem "Programming" a druhý určuje, jak bude vypadat nová hodnota.

Výpis 4.13: Nahrazení dokumentu.

```
db.collection.update(
  { course: "Programming" },
  {
    course: "Programming␣for␣beginners"
    credits: 5,
    teacher: "John␣Doe"
  }
)
```

Změnové operátory

Dalším způsobem, jak změnit hodnoty dokumentů, je s použitím některého ze změnových operátorů. V takovém případě má druhý parametr formát 4.14.

Výpis 4.14: Formát objektu pro změnu dokumentu s použitím změnových operátorů.

```
{
  <update operator>:
  {
    <field1>: <value1>,
    <field2>: <value2>,
    ...
  },
  <update operator>:
  {
    ...
  }
}
```

Změnových operátorů je velké množství a jejich kompletní seznam je k nalezení v oficiální dokumentaci [19]. Některé z těch nejdůležitějších jsou však uvedeny v tabulce 4.4.

Použití operátoru $set je ilustrováno na příkladě 4.15.

Výpis 4.15: Změna dokumentu s použitím operátoru set.

```
db.collection.update(
  { course: "Programming" },
  $set: {course: "Programming␣for␣beginners" }
)
```

Tab. 4.4: Vybrané změnové operátory MongoDB

Zápis	**Popis**
$set	Nastaví hodnotu pole
$setOrInsert	Nastaví hodnotu pole nebo vloží nové
$unset	Smaže pole z dokumentu
$addToSet	Přidá prvek do pole, pokud tam doposud není
$push	Přidá prvek do pole
$pull	Smaže z pole prvky na základě filtračních parametrů

4.4.4 Mazání

K mazání dokumentů slouží metoda *Remove*, která akceptuje filtrační parametry stejné jako u vyhledávání a změny dat.

4.5 Pokročilá funkcionalita

4.5.1 Validátory

MongoDB ve svém výchozím nastavení neprovádí žádnou validaci formátu vkládaných dokumentů. K tomuto úkolu slouží tzv. validátor, který je navázaný na kolekci. Validátor specifikuje JSON schéma, kterému musí vkládané dokumenty odpovídat. Každá kolekce může mít maximálně jeden validátor. U validátoru jsou důležitá dvě nastavení. Tím prvním je akce (validationAction), která se má provést, pokud vkládaný dokument není validní:

- **Chyba** (error)- dokument není uložen.
- **Varování** (warning) - dokument je uložen, ale uživateli je vypsáno varování [19].

Druhým důležitým nastavením u validátorů je validační level (validationLevel), který má tři možnosti:

- **Vypnuto** (off) - validátor je přítomen, ale validace samotná je vypnutá.
- **Striktní** (strict) - validace se uplatňuje na všechny dokumenty.
- **Mírná** (moderate) - validace se uplatňuje pouze na nově vzniklé dokumenty a staré, které již byly ve validním formátu. Neuplatňuje se při změně již existujících nevalidních dokumentů [19].

Validátory pomáhají udržet schéma v určitém stavu a mohou být nápomocné k nalezení chyb v aplikaci, která do databáze vkládá data. Nevýhodou validátorů je, že zpomalují zápis, rychlost čtení validátory ovlivněna není [9].

4.5.2 Indexy

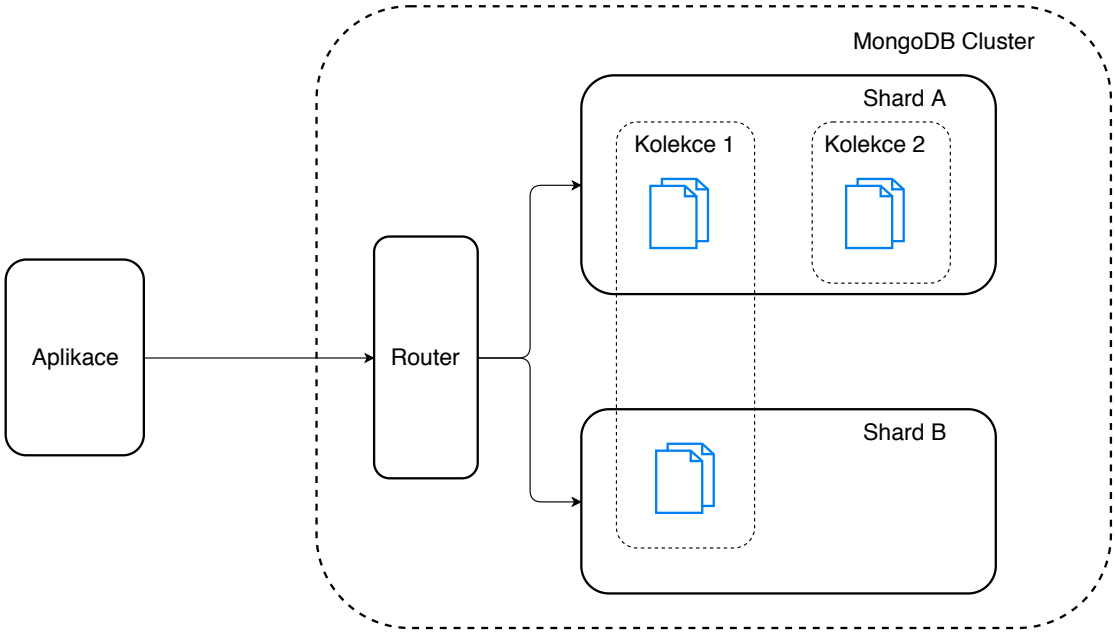
Indexy u MongoDB slouží ke stejnému účelu jako u databází relačních, a to ke zrychlení vyhledávání a řazení dat. Indexy jsou vždy vázany na kolekci. Indexů existuje celá řada

- **Pro jednu položku** (single field) vytváří index nad jedním konkrétním polem. Tento typ indexu je vhodné použít, pokud často provádím vyhledávání nebo řazení právě nad jedním polem.
- **Složené** (compound) jsou indexy nad více poli najednou. Používají se pro zrychlení řazení pomocí více než jednoho pole.
- **Indexy pro pole** (multikey) indexují pole. Tento index na pozadí vytvoří více indexů - pro každý typ objektu v poli vlastní. Tyto indexy se používají pro zrychlení vyhledávání v poli.
- **Textové** (text) zrychlují textové vyhledávání nad jednotlivými položkami v kolekci.
- **Geolokační** (geospatial) jsou speciálním typem indexu usnadňujícím práci se souřadnicemi [19].

4.5.3 Škálovatelnost

MongoDB podporuje horizontální škálovatelnost. To znamená, že databáze nemusí běžet na jediném stroji, ale na více, což umožňuje rozdělení zátěže a v konečném důsledku větší prostupnost dat a větší množství dotazů, které je databáze schopná zvládnout. Souboru počítačů, které databáze používá, se označuje jako cluster, a jednotlivé počítače v něm jako shardy [9].

Pokud databáze běží v clusteru, je možné u kolekce nastavit, zda-li má škálovat nebo ne. Pokud u kolekce není škálování/shardování zapnuto, její data se ukládají na primárním shardu, kde se zároveň provádějí operace nad ní. Pro rozložení zátěže u kolekce také označované jako sharding je nejprve nutné vytvořit tzv. sharding key, které slouží k určení shardu, na který se data uloží. Situace je ilustrována na obr. 4.2 [19].



Obr. 4.2: MongoDB cluster

4.6 Limity databáze

Dokumentace MongoDB ve verzi 4.2 uvádí některá omezení pro dokumenty a kolekce:

- **Maximální velikost dokumentu** je 16 MB.
- **Maximální počet indexů** na kolekci je 64.
- **Složený index** může obsahovat maximálně 31 polí.
- **Maximální počet dokumentů** v jedné kolekci na jednom shardu je 2^{32}.
- **Řazení dokumentů** je omezeno celkovou velikostí všech dokumentů, které se mají seřadit. Nemůže přesáhnout 32 Mb.
- **Hromadný zápis** je omezen na 100000 dokumentů v jedné dávce. [19].

5 Návrh databáze

5.1 Požadavky

Před samotným návrhem databáze je nutné vyjmenovat požadavky a očekávání, která by měla splňovat.

5.1.1 Rychlost zápisu

Prvním zcela jasným požadavkem je rychlost zápisu. GPON sítě jsou schopny nabídnout 2,488 Gb/s v sestupném směru, lze tedy očekávat velmi vysokou zátěž. Celkový počet GEM rámců za sekundu se velmi těžko odhaduje, protože toto číslo závisí na velikosti přenášených dat. Je však poměrně snadné určit spodní hranici. GEM rámec může přenášet maximálně 4095 bajtů dat, a tudíž se dá očekávat minimálně kolem 76 tisíc GEM rámců za sekundu. Za předpokladu, že by jeden GEM rámec nesl polovinu maximální velikosti, musí databáze zvládat dvojnásobek minimálního počtu tj. 152 GEM rámců za sekundu.

5.1.2 Datový formát

Databáze byla navrhována tak, aby mohla být použita a integrována s již existující aplikací, která obstarává sběr dat. Tato aplikace již produkuje data v JSON formátu a bylo by vhodné zachovat jeho strukturu a pojmenování jednotlivých polí. Použití jiného formátu je možné, avšak konverze z jednoho JSON formátu do druhého by přidávala latenci a znepřehlednila by chování systémů nekonzistecí dat mezi aplikacemi.

Aplikace pro sběr dat používá pro formát GTC hlavičky formát 5.1, kde *BWmap* obsahuje pole alokačních struktur ve formátu 5.2.

Výpis 5.1: JSON struktura GTC rámce.

```
{
  "Psync": 123,
  "Identification":
  {
    "FEC": True,
    "Reserved": False,
    "SuperframeCounter": 994141217
  },
  "PLOAMdownstream":
  {
    "ONUid": 255,
    "MessageID": 11,
    "Data": "AAAAAAAAAAAAA==",
    "CRC": 158
  },
  "BIP": 196,
  "Plend": [
  {
    "Blen": 6,
    "Alen": 0,
    "CRC": 245
  }, {
    "Blen": 6,
    "Alen": 0,
    "CRC": 245
  }
  ],
  "Bwmap": [...]
}
```

Výpis 5.2: JSON alokační struktury BWmap pole.

```
  {
    "AllocID": 0,
    "Flags": 0,
    "StartTime": 0,
    "StopTime": 79,
    "CRC": 234
  }
```

Je možné si povšimnout, že v GTC formátu 5.1 je *Plend* pole zduplikováno. Toto

je standardní chování protokolu, který toto pole vkládá dvakrát nebo třikrát jako protichybovou prevenci.

Formát GEM hlavičky nebyl předem specifikován, a proto je předmětem návrhu.

5.1.3 Čtení a rekonstrukce hlaviček

Z dat v databázi by v co nejkratším čase mělo jít sestavit původní hlavičky pro další zpracování. Pole *BWmap* však pro následné zpracování není potřeba, a proto nemusí být načteno.

5.1.4 Nepožadováno

Vyhledávání

Co se týče rychlosti vyhledávání nebo případných častých dotazů nad databází, tak doposud žádné takové požadavky nejsou známy.

Validace

Od databáze se také neočekává žádná validace ukládaných dat. Tu, jak tomu již u NoSQL databází bývá, musí zajistit aplikace, která ji používá.

5.2 Schéma

5.2.1 Kolekce

Z výše uvedench požadavků vychází, že je vhodné GPON komunikaci uložit do co nejmenšího počtu kolekcí. A to z toho důvodu, že čím více kolekcí bude databáze mít, tím více práce bude mít aplikace, která z ní bude rekonsturovat hlavičky. Dalším důvodem je, že rozdělení na více kolekcí znamená více ukládacích příkazů, což veškerou práci zpomaluje.

Dvě základní struktury, které v GPON komunikaci nalezneme, jsou GTC a GEM rámce. Ukazuje se jako vhodné vytvořit alespoň kolekce pro tyto objekty a to ze dvou důvodů: Jednak jsou v samotné komunikaci odděleny daty a nejdříve dochází k přijetí GTC hlavičky a až potom GEM hlavičky. Aplikace, která databázi používá, může proto paralelně ukládat GTC hlavičky a číst nebo zpracovávat GEM rámce. Druhým důvodem proč mít GTC a GEM v samostatných kolekcích je jejich odlišný datový formát.

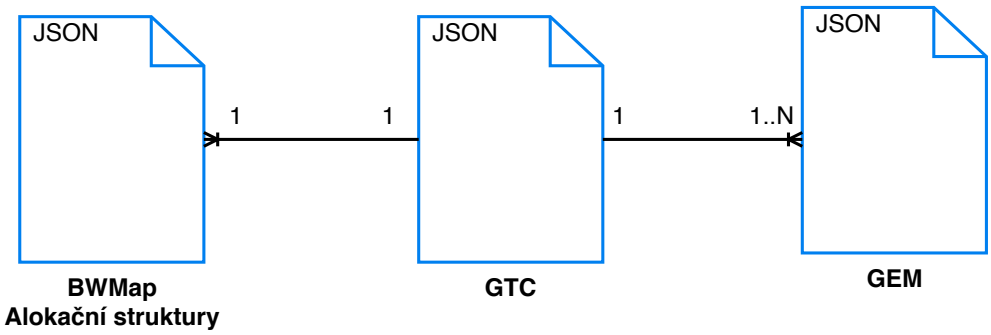
Další strukturou, která může být kandidátem pro vlastní kolekci, je pole *BWmap*. Důvodem k oddělení sturktury *BWmap* je její nepotřebnost pro rekonstrukci

hlaviček diskutovaná v kapitole 5.1.3. Oddělení ostatních polí a struktury jako např.*PLOAMDownstream* nebo *Plend* nepřináší žádné výhody.

Ukazuje se tedy, že by MongoDB měla obsahovat následující tři kolekce:

- GTC,
- GEM a
- BWmap.

Logické závistlosti mezi těmito kolekcemi jsou vidět na obr. 5.1.



Obr. 5.1: Kolekce pro uchování GPON komunikace

GTC kolekce

Schéma GTC kolekce vychází z formátu v požadavcích 5.1, ale bude z něho odebrána bandwidth mapa, která bude v samostatné kolekci. Finální schéma dat vkládaných do této kolekce je vidět na 5.4.

Výpis 5.3: JSON struktura GTC kolekce.

```
{
  "Psync": <int>,
  "Identification":
  {
    "FEC": <boolean>,
    "Reserved": <boolean>,
    "SuperframeCounter": <int>
  },
  "PLOAMdownstream":
  {
    "ONUid": <int>,
    "MessageID": <int>,
    "Data": <string nebo object>,
    "CRC": <int>
  },
  "BIP": <int>,
  "Plend": [
    {
      "Blen": <int>,
      "Alen": <int>,
      "CRC": <int>
    }, {
      "Blen": <int>,
      "Alen": <int>,
      "CRC": <int>
    }
  ]
}
```

GEM kolekce

Schéma GEM kolekce bylo navrženo tak, aby odpovídalo struktuře GEM hlavičky, je však nutné přidat pole *GtcId* k vytvoření logické vazby na GTC kolekci.

Výpis 5.4: JSON struktura GEM kolekce.

```
{
  "GtcId": <ObjectId>,
  "PLI": <int>
  "PortId": <int>,
  "HEC": <int>,
  "PTI":
  {
    "LastFragment": <bool>,
    "UserData": <bool>
  }
}
```

Za povšimnutí stojí, že třetí bit z pole *PTI* ve schématu není obsažen, protože se momentálně nepoužívá.

BWmap kolekce

Struktura dat v BWmap kolekci opět vychází z požadavků 5.1 a stejně jako u GEM kolekce, i zde je přidáno pole *GtcId*.

Výpis 5.5: JSON struktura BWmap kolekce.

```
{
  "GtcId": <ObjectId>,
  "Bwmap": [
  {
    "AllocID": <int>,
    "Flags": <int>,
    "StartTime": <int>,
    "StopTime": <int>,
    "CRC": <int>
   }, {
    "AllocID": <int>,
    "Flags": <int>,
    "StartTime": <int>,
    "StopTime": <int>,
    "CRC": <int>
  },
  ...
  ]
}
```

5.2.2 Validátory

Z důvodu zpomalování databáze nejsou validátory součástí schématu, avšak byly při návrhu zvažovány a GTC validátor je uveden v příloze A.

6 Práce s databází

Samotná databáze by bez aplikace, která by ji používala, nebyla užitečná. Následující kapitola je zaměřená na popis toho, jak s databází pracovat z aplikačního zdrojového kódu. MongoDB nabízí řadu ovladačů pro různé programovcí jazyky včetně C, C++, C#, Javy, Rustu, Pythonu a řady dalších. Tato práce popisuje, jak s navrženou databází pracovat v programovacích jazycích Python a C# [19].

Python byl vybrán ze dvou důvodů. Zaprvé se jedná o jeden z nejoblíbenějších backendových[1] jazyků současnosti a zároveň je znám svou přímočarostí umožňující rychlé prototypování [28].

Jako další programovací jazyk byl vybrán C#. C# byl zvolen, protože tato práce částečně vychází z C# modulu určeného pro zpracování GPON komunikace publikovaného v [18]. Pokud by mělo dojít k integraci databáze s tímto modulem, je vhodnější, aby obě části byly implementovány ve stejném jazyce. Při použití stejného jazyka je možné sdílet části kódu a nedochází ke komplikacím během integrace odlišných technologií [18].

6.1 Vzor Repository

K usnadnění práce s databází je využit softwarový vzor Repository, který slouží k zapouzdření logiky pro persistenci dat a práci s datovým úložištěm. Tento vzor byl použit u obou implementací v jazyku Python i u jazyka C#.

Jednotlivé implementace se však liší pojmenováním metod a datovými typy na vstupu a výstupu metod, avšak účel jednotlivých tříd a jejich metod je vždy stejný. Pojmenování metod se liší z důvodu dodržení konvencí daného jazyka. Metoda ukládající jeden prvek byla např. v implementaci pro Python pojmenována *insert_one* a pro C# *InsertOne*. Kompletní přehled názvů metod je v tabulce 6.1. Datové typy se liší z důvodu použití různých ovladačů a knihoven.

6.1.1 Rodičovská třída MongoRepository

K obecné práci s libovolnou kolekcí slouží třída *MongoRepository*. Třída *MongoRepository* obsahuje následující metody:

- **insert_one**, jež vloží do kolekce jeden prvek a vrátí jeho jedinečný identifikátor,
- **insert_many** k hromadnému uložení více záznamů najednou,
- **update_one** sloužící ke změně jednoho záznamu v kolekci,

[1] Backend je označení pro logiku na pozadí, která většinou běží na serveru a nesouvisí s uživatelským m.

Tab. 6.1: Srovnání pojmenování metod repozitářů pro různé implementace

Python - synchronní	**Python - asynchronní**	**C#**
insert_one	insert_one_async	InsertOneAsync
insert_many	insert_many_async	InsertManyAsync
update_one	update_one_async	UpdateOneAsync
update_many	update_many_async	UpdateManyAsync
delete_one	delete_one_async	DeleteOneAsync
delete_many	delete_many_async	DeleteManyAsync
get_by_id	get_by_id_async	GetByIdAsync
find	find_async	FindAsync
get_by_gtc_id	get_by_gtc_id_async	GetByGtcIdAsync

- **update_many** k změně více záznamů najednou,
- **delete_one** smaže jeden záznam na základě jeho identifikátoru,
- **delete_many** smaže více záznamů,
- **get_by_id** načte jeden záznam na základě jeho identifikátoru a
- **find** sloužící k vyhledávání záznamů.

6.1.2 Specifické třídy pro kolekce

Pro práci s třemi navrženými kolekcemi byly vytvořeny následující specifické repozitáře, které dědí[2] třídu *MongoRepository*:

- **GtcRepository**,
- **GemRepository** a
- **BwMapRepository**.

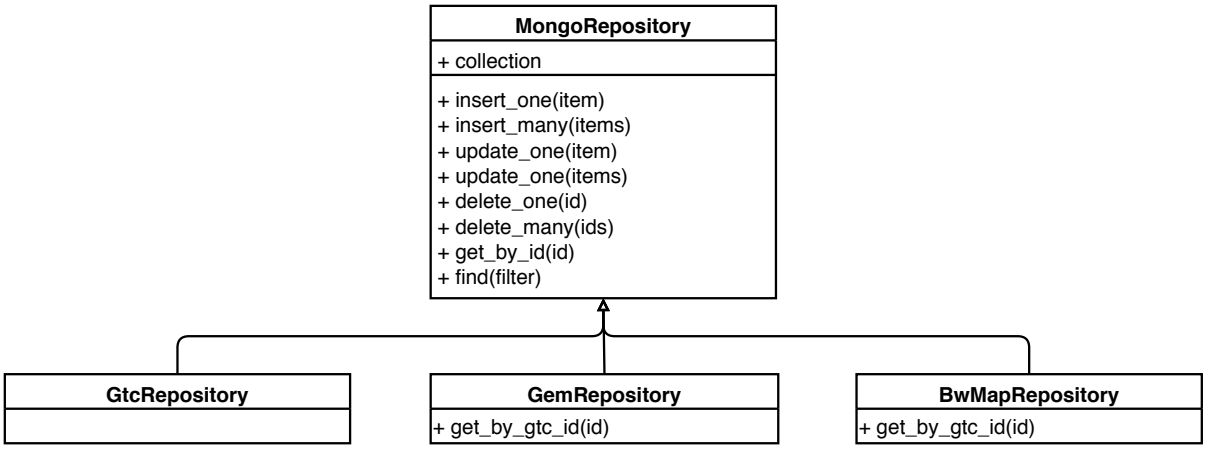
Tyto třídy již jasně definují, se kterou kolekcí se pracuje. *GemRepository* a *BwMapRepository* navíc obsahují metodu *find_by_gtc_id*, které umožňují najít příslušné záznamy pro konkrétní GTC hlavičku.

Všechny třídy jsou znázorněny na diagramu tříd 6.1.

6.2 Python a C# jako programovací jazyky

Python i C# se velmi často používají jako jazyky pro tvorbu logiky na pozadí, avšak i přes podobné použití se v mnohém liší. Tato kapitola stručně popisuje základní rozdíly mezi těmito jazyky.

[2]Dědičnost je jedním ze základních konceptů objektově orientovaného programování.



Obr. 6.1: Obecný diagram tříd repozitářů

6.2.1 Staticky a dynamicky typovaný jazyk

Jedním z největších rozdílů mezi oběma jazyky je jejich typový systém. Python je dynamicky typovaný jazyk. To znamená, že proměnné uložené v paměti mohou změnit svůj datový typ, proto u deklarace proměnných není potřeba typ specifikovat. Ukázka dynamického typování v jazyce Python je vidět v příkladu 6.1 [23, 24].

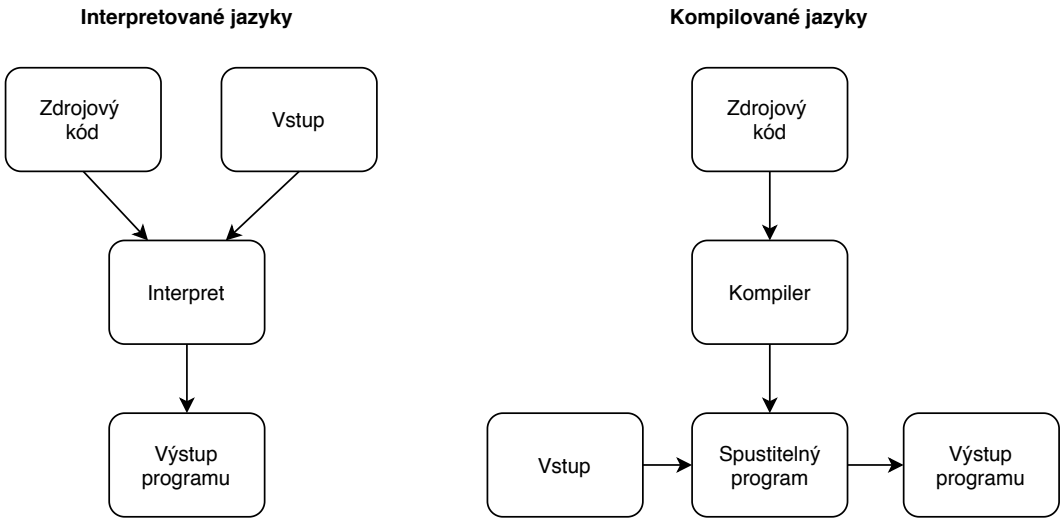
Výpis 6.1: Ukázka dynmického typování v jazyce Python

```
promenna = 5          # Priradime promenne cislo
promenna = [1, 2, 3] # Priradime promenne kolekci
promenna = "text"     # Priradime promenne text
```

C# je naopak staticky typovaným jazykem. Již u deklarace proměnných je nutné specifikovat její typ, který se bez akce programátora nemůže změnit.[24, 25]

6.2.2 Interpretovaný a kompilovaný jazyk

Dalším podstatným rozdílem je způsob exekuce zdrojového kódu. Aby bylo možné spustit libovolný skript nebo aplikaci, je nutné jeho zdrojový kód přeložit do instrukcí procesoru. Python spadá do kategorie interpretovaných jazyků. Jeho kód se začne překládat (interpretovat) až v okamžiku spuštění. Oproti tomu C# je kompilovaný jazyk. Aby bylo možné spustit aplikaci vytvořenou v tomto jazyce, musí nejprve proběhnout proces kompilace a až zkompilovaný kód je možné spustit. Situace je znázorněna na obrázku 6.2 [23, 24].



Obr. 6.2: Rozdíl mezi interpretovanými a kompilovanými jazyky.

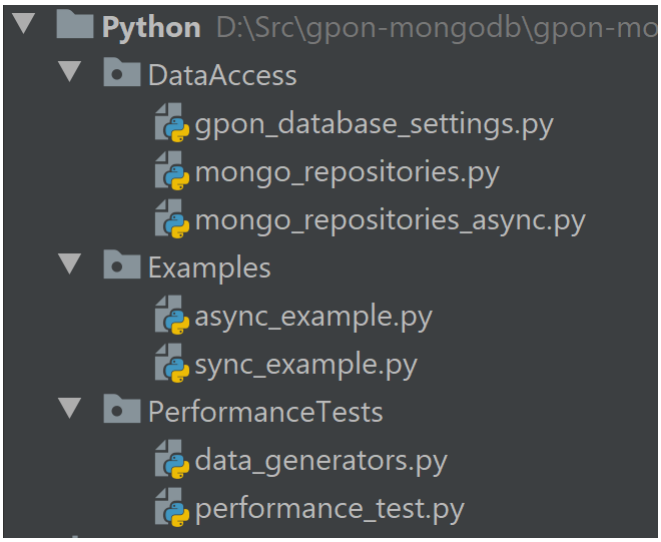
Výhoda interpretovaných jazyků spočítá v odstranění času ke kompilaci, která je u jazyka C# nutná i při změně jediného řádku kódu. Naopak kompilované jazyky již z podstaty věci bývají rychlejší.

U obou jazyků toto rozdělení není zcela striknì, neboť se u Pythonu můžeme setkat s tzv. just-in-time kompilerem, který umožňuje provést kompilaci během prvního spuštění a zapamatovat si její výsledek. U C# se ve většině případů neprovádí kompilace přímo, ale dochází nejprve k překladu do mezijazyka Common intermediate language, který je až poté kompilován [23, 24].

6.3 Struktura řešení

Struktura řešení v jazyce Python je k vidění na obrázku 6.3 a sestává ze tří hlavních složek:

- **DataAccess**, který obsahuje repozitory třídy a pomocnou třídu pro uchování jmen kolekcí a jména databáze,
- **Examples** s příklady použití repozitářů a
- **PerformanceTests** s kódem pro výkonostní testování.



Obr. 6.3: Struktura Python řešení.

6.4 Práce v jazyce Python

Pro práci s MongoDB databází v programovacím jazyce Python může být využit jeden z následujících ovladačů, které jsou veřejně ke stažení:

- **PyMongo**, které umožňuje synchronní práci s databází, a
- **Motor**, který je jeho nástavbou a umožňuje asynchronní práci [19].

Vzhledem k faktu, že pro synchronní a asynchronní práci jsou nutné dvě různé knihovny a tedy i sady tříd, je i implementace takto oddělena.

6.4.1 Reprezentace ukládaných dat

Oba dva ovladače k reprezentaci dokumentů, které se mají uložit do databáze, používají zanořené slovníky. Ukázka GTC hlavičky definované pomocí zanořených slovníků je uvedena v příkladu 6.2, GEM v příkladu 6.3 a BW mapy 6.4.

Výpis 6.2: Vytvoření objektu v Pythonu reprezentující GTC hlavičku

```
gtc_header = {
  "Psync": 123,
  "Identification": {
    "FEC": True,
    "Reserved": False,
    "SuperframeCounter": 994141217
  },
  "PLOAMdownstream": {
    "ONUid␣": 255,
    "MessageID␣": 11,
    "Data": "",
    "CRC": 158
  },
  "BIP": 196,
  "Plend": [{
    "Blen": 6,
    "Alen": 0,
    "CRC": 245
  }, {
    "Blen": 6,
    "Alen": 0,
    "CRC": 245
  }]
}
```

Výpis 6.3: Vytvoření objektu v Pythonu reprezentující GEM hlavičku

```
gem = {
  "GtcId": 1,
  "PLI": 1,
  "PortId": 1,
  "HEC": 1,
  "PTI": {
    "LastFragment": False
  }
}
```

Výpis 6.4: Vytvoření objektu v Pythonu reprezentující BW mapu

```
gem = {
  "GtcId": 1,
  "PLI": 1,
  "PortId": 1,
  "HEC": 1,
  "PTI": {
    "LastFragment": False
  }
}
```

6.4.2 Ovladač PyMongo a synchronní komunikace

Ovladač PyMongo

Pro synchronní práci s databází je nutné nainstalovat ovladač obsažený v balíčku PyMongo. Toho můžeme docílit příkazem 6.5 [26].

Výpis 6.5: Instalace PyMongo balíčku

```
python -m pip install pymongo
```

K použití objektů z tohoto balíčku je nutné ho naimportovat. Import veškerých tříd z tohoto balíčku provedeme příkazem 6.6 [26].

Výpis 6.6: Import tříd z PyMongo balíčku

```
from pymongo import *
```

Připojení k databázi

Samotné připojení k databázi zajišťuje třída *MongoClient*. Konstruktor třídy *MongoClient* má celou řadu parametrů souvisejících s připojením k databázi včetně následujících:

- **host** - IP adresa nebo doménové jméno databázového serveru,
- **port** - číslo portu, na kterém databáze běží,
- **username** - uživatelské jméno použité k přihlášení do databáze,
- **password** - heslo pro přihlášení do databáze,
- a další [26].

Ukázka vytvoření instance třídy *MongoClient* je uvedena v příkladu 6.7. Tato třída již umožňuje přistupovat k jednotlivým databázím a jejich kolekcím běžícím na MongoDB serveru.

Výpis 6.7: Vytvoření databázového klienta knihovny PyMongo

```
client = MongoClient("localhost", 27017)
```

Repository třídy

Třída *MongoRepository* popsaná v kapitole 6.1 ve svém konstruktoru příjmá kolekci typu *pymongo.collection*. Specifické repozitáře *GtcRepository*, *GemRepository* a *BwMapRepository* poskytují nápovědy typů tzv. type hints[3]. Kompletní kód těchto repozitářů je uveden v příloze B.

Použití kódu

Vytvoření jednoho záznamu v GTC kolekci je znázorněno v příkladu 6.8. Po vložení záznamu do GTC kolekce je vrácen jedinečný identifikátor nově vytvořeného záznamu *gtc_id*. Vložení jednoho záznamu do kolekce BwMap a GEM je vidět v příkladu 6.9. Oba příklady využívají metodu pro vložení jednoho dokumentu *insert_one*.

Výpis 6.8: Vytvoření GTC záznamu pomocí Python repozitáře

```
from pymongo import MongoClient
from mongo_repositories import GtcRepository

# Klient k pripojeni k databazi
client = MongoClient("localhost", 27017)
# Repozitar pro praci s GTC kolekci
gtc_repository = GtcRepository(client)

gtc_header = {
  # Data pro GTC hlavicku
}

gtc_id = gtc_repository.insert_one(gtc_header)
```

[3]Přesné typy definovat není možné, neboť Python je dynamicky typovaný jazyk.

Výpis 6.9: Vytvoření GEM a BwMap záznamů pomocí python repozitářů

```
gem_repository = GemRepository(client)
bwmap_repository = BwMapRepository(client)

bandwidth_map = {
  # Data pro bandwith mapu
}

bwmap_repository.insert_one(bandwidth_map)

gem = {
  # Data pro GEM hlavicku
}

gem_repository.insert_one(gem)
```

Úprava dat je možná pomocí funkcí *update_one* nebo *update_many*. V příkladu 6.10 je ukázáno načtení GEM objektů patřících do jednoho GTC a jejich následná změna.

Výpis 6.10: Úprava GEM dokumentů patřící k jednomu GTC

```
gtc_id = ObjectId("5ed45343a3c8d93d128e4105")

gems = gem_repository.get_by_gtc_id(gtc_id)
for gem in gems:
  gem["PortId"] = 42

gem_repository.update_many(gems)
```

Obdobně k mazání dokumentů slouží funkce *delete_one* nebo *delete_many*. Příklad 6.11 ukazuje, jak tyto funkce použít k smazání všech dokumentů patřících k jednomu GTC.

Výpis 6.11: Mazání všech dokumentů patřícího do jednoho GTC

```
gtc_id = ObjectId("5ed4596bf011f7d8a977a8f8")

# Nalezeni GEM prislusicich jednomu GTC
gems = gem_repository.get_by_gtc_id(gtc_id)
gem_ids = list(map(lambda x: x["_id"], gems))

# Nalezeni bandwidth mapy
bandwidth_map = bwmap_repository.get_by_gtc_id(gtc_id)

# Smazani bandwidth mapy
bwmap_repository.delete_by_id(bandwidth_map["_id"])

# Hromadne smazani GEM dokumentu
gem_repository.delete_by_ids(gem_ids)

# Smazani GTC dokumentu
gtc_repository.delete_by_id(gtc_id)
```

6.4.3 Ovladač Motor a asynchronní komunikace

Asynchronní komunikace je takový druh komunikace, u něhož není nutné čekat na dokončení každé operace, která jde do databáze před zahájením dalších instrukcí programu nebo dokonce jiné komunikace se stejnou databází.

Ovladač Motor

Asynchronní komunikace s MongoDB zajišťuje knihovna Motor. Motor samotný nezajišťuje provádění asynchronních operací, místo toho je kompatibilní s dvěma knihovnami určenými k tomuto účelu, kterými jsou: Asyncio a Tornado. Knihovna Asyncio umožňuje vytvářet asynchronní kód s použitím klíčových slov *async* a *await*. Na této knihovně je postavené velké množství jiných Python knihoven. Tornado je webový framework určený pro síťovou komunikaci [19, 27].

Vzhledem k faktu, že většina testování komunikace s databází se provádí na stejném počítači, kde se databáze nachází, a to z toho důvodu, aby bylo dosaženo minimální latence a maximálního výkonu, tak jako asynchronní knihovna byla zvolena asyncio.

Instalace knihovny Motor je možné provést příkazovou řádkou 6.12. Aby mohla být tato knihovna použita, je nutné ji naimportovat spolu s již uvedenou asyncio, jak je znázorněno v příkladu 6.13.

Výpis 6.12: Instalace balíčku Motor

```
python -m pip install motor
```

Výpis 6.13: Import balíčku Motor spolu s asyncio

```
import asyncio
import motor.motor_asyncio
```

Repository třídy

Pro asynchronní komunikaci byla vytvořena sada repository tříd obdobných těm z diagramu 6.1. Tyto třídy na pozadí nepoužívají ovladač PyMongo nýbrž Motor. Aby se odlišily od synchronních, nesou příponu "Async". Všechny metody těchto tříd jsou označeny klíčovým slovem *async*. Jedná se tedy o tzv. korutiny, které vrací návratovou hodnotu typu awaitable. Kompletní kód těchto tříd je uveden v příloze C [29].

Použití kódu

Asynchronní repository třídy se používají obdobně jako sychronní. Pouze s tím rozdílem, že pokud je zavolána kterákoliv metoda z asynchronních repozitářů, nečeká se na dokončení této operace. Namísto toho je vrácena tzv. awaitable hodnota. Aby program vyčkal na dokončení dané operace, je nutné použít klíčové slovo *await*, a nebo využít čekací smyčku z knihovny Asyncio. Použití knihovny Asyncio v kombinaci s asynchronními repozitáři je znázorněno na příkladu 6.14.

Výpis 6.14: Použití asynchronních repozitářů

```
from motor.motor_asyncio import AsyncIOMotorClient

# Klient k pripojeni k databazi
client = AsyncIOMotorClient('localhost', 27017)

# Vytvoreni repozitaru
gtc_repository = GtcRepositoryAsync(client)
gem_repository = GemRepositoryAsync(client)

gtc_header = {
  # Vytvoreni GTC hlavicky
}

loop = asyncio.get_event_loop()

gtc_insert_task = gtc_repository\
          .insert_one_async(gtc_header)

# Vyckame na dokonceni
gtc_id = loop.run_until_complete(gtc_insert_task)

gem = {
  # Vytvoreni GEM hlavicky
}

gem_insert_task = gem_repository.insert_one_async(gem)

# Vyckame na dokonceni
loop.run_until_complete(gem_insert_task)

loop.stop()
```

6.5 Práce v jazyce C#

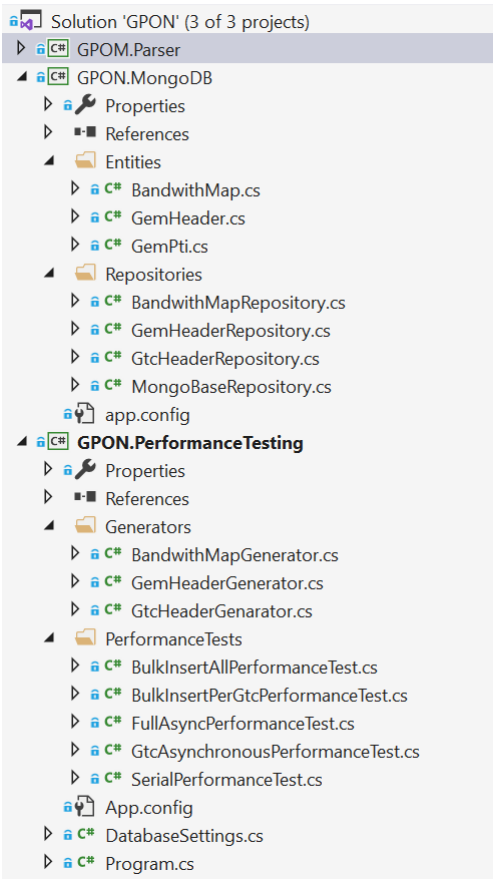
Následující kapitola popisuje vytvořené C# knihovny uzpůsobené pro práci s navrženou databází. K vytvoření knihoven byl použit C# ve verzi 8.0[4] a soubor standardních knihoven .NET ve verzi 4.7.2

[4]V době tvorby práce se jednalo o nejnovější verzi.

6.5.1 Struktura řešení

Struktura řešení v jazyce C# je k vidění na obrázku 6.4 a sestává ze tří projektů:

- **GPON.MongoDB**, který obsahuje repozitory třídy a pomocné entity pro práci databází,
- **GPON.PerformanceTesting** se zdrojovým kódem k výkonnostnímu testování z kapitoly 7 včetně generátorů dat,
- **GPON.Parser**, který obsahuje zdrojový kód z práce [18].



Obr. 6.4: Struktura C# řešení.

6.5.2 Ovladač MongoDB.Driver

Pro práci s MongoDB databází v jazyce C# je v balíčkovacím systému Nuget dostupný ovladač *MongoDB.Driver*. Tento ovladač je již asynchronní, přičemž k asynchronnosti využívá stávající async-await syntaxi C# a třídu *System.Threading.Task*, která je součástí standardní sady C# knihoven .NET.

Tuto knihovnu je možné přidat několika způsoby. První možností je použít grafické rozhraní Visual Studio[5] pro správu projektových referencí. Další možností je instalace promocí příkazové řádky 6.15, která musí být spuštěna z adresáře projektu, který ji bude používat.

Výpis 6.15: Příkazová řádka k přidání C# ovladače MongoDB.Driver

```
dotnet add package MongoDB.Driver
```

K použití tříd z tohoto balíčku je nutné ho naimportovat pomocí příkazu 6.16.

Výpis 6.16: Import tříd z PyMongo balíčku

```
using MongoDB.Driver;
```

6.5.3 Reprezentace dat

V C# se k reprezentaci dat nepoužívá struktura slovníků, ale třídy. Je tedy nezbytné tyto třídy vytvořit. Část tříd je přepoužita z projektu *GPON.Parser*, některé musely být vytvořeny a jsou obsaženy ve složce *Entities* v projektu *GPON.MongoDB*.

Ukázka vytvoření GTC hlavičky je k vidění na příkladu 6.17, GEM hlavičky na příkladu 6.18 a bandwidth mapy obsahující n položek na příkladu 6.19.

[5]Visual Studio je nejčastější nástroj pro tvorbu C# aplikací.

Výpis 6.17: Vytvoření objektu v C# reprezentující GTC hlavičku

```
var gtcHeader = new GPONFrame
        {
          PCBdownstream = new GPONFrame.PCBd
          {
            Psync = 123,
            BIP 1= 196,
            Identification = new GPONFrame.PCBd.Ident
              {
                FEC = 1,
                Reserved = 1,
                SuperframeCounter = 994141217
              },
            Plend = new GPONFrame.PCBd.PLend[]
            {
              new GPONFrame.PCBd.PLend
              {
                Blen = 6,
                Alen = 0,
                CRC = 245
              }
            }
          }
        }
```

Výpis 6.18: Vytvoření objektu v C# reprezentující GEM hlavičku

```
  var gemHeader = new GemHeader
      {
        GtcId = 42,
        PortId = 1,
        HEC = 1,
        PTI = new GemPti
        {
          LastFragment = false,
          UserData = true
        }
      };
```

Výpis 6.19: Vytvoření objektu v C# reprezentující BW mapu

```
var bwMapItems = new BWmap[n];
for(int i = 0; i < n; i++)
{
  bwMapItems[i] = new BWmap
  {
    AllocID = 1,
    Flags = 1,
    StartTime = 1,
    StopTime = 1
  };
}

var bwMap = new BandwithMap
  {
    GtcId = 42,
    Maps = bwMaps
  };
```

6.5.4 Použití kódu

Připojení k databázi

I v jazyce C# je nutné nejprve vytvořit klienta pro připojení a následně získat databázi, se kterou je potřeba pracovat. Knihovna *MongoDB.Driver* za tímto účelem poskytuje třídu *MongoClient* s metodou *GetDatabase*. Vytvoření klienta a získání databáze je vidět v příkladu 6.20.

Výpis 6.20: Vytvoření klienta pro GEM dokumentů patřící k jednomu GTC

```
var mongoClient = new MongoClient(
          @"mongodb://localhost:27017");
var gponDatabase = mongoClient.GetDatabase("gpon");
```

Informace o nastavení databáze jako je její jméno, názvy kolekcí a informace k připojení je možné uložit do aplikačního konfirugračního souboru v XML formátu v sekci *appSettings*. Ukázka takového nastavení je uvedena v příkladu 6.21. Tato konfigurace je v zdrojovém kódu dostupná pomocí statické třídy *DatabaseSettings*.

Výpis 6.21: XML konfigurace pro připojení k databázi v jazyce C#

```
<appSettings>
  <add key="ConnectionString"
     value="mongodb://localhost:27017" />
  <add key="DatabaseName" value="gpon" />
  <add key="GtcCollection" value="gtc" />
  <add key="GemCollection" value="gem" />
  <add key="BwMapCollection" value="bwmap" />
</appSettings>
```

Práce s repozitáři

Repozitáře pro GTC, GEM a bandwidth map mají vždy dva konstruktory:

- **bez parametrů**, který využívá nastavení ze třídy *DatabaseSettings*,
- **s parametry** umožňující specifikovat databázi a jméno kolekce.

C# repozitáře byly navrženy tak, aby co nejvíce odpovídaly těm v Pythonu. Jejich použití je proto obdobné. Jediným výrazným rozdílem je fakt, že repozitáře jsou automaticky asynchronní. Za účelem synchronního použití je nutné na dokončení operací počkat pomocí klíčového slovat *await*. Ukázka sériového vytvoření GTC dokumentu je uvedena v příkladu 6.22 a vytvoření GEM a bandwidth mapy v příkladu 6.23.

Výpis 6.22: Vytvoření GTC dokumentu v jazyce C#

```
using GPON.MongoDB;
using GPON.MongoDB.Entities;
using GPON.MongoDB.Repositories;
using GPON_parser;
using System.Threading.Tasks;

namespace GPON.Examples
{
  public class SynchronousInsert
  {
    public async Task InsertGtc()
    {
      var gemRepository = new GemHeaderRepository();
      var gtcHeader = new GPONFrame
      {
         // Obsah GTC
      };

      var gtcId = await gtcRepository
             .InsertOneAsync(gtcHeader);

    }
  }
}
```

Výpis 6.23: Vytvoření GEM a bandwidth mapy v jazyce C#

```
using GPON.MongoDB;
using GPON.MongoDB.Entities;
using GPON.MongoDB.Repositories;
using GPON_parser;
using System.Threading.Tasks;

namespace GPON.Examples
{
  public class SynchronousInsert
  {
    public async Task InsertGemAndBwMap()
    {
      //  Muze navazovat na vytvoreni GTC
      string gtcId = "5dfe90a06f38bf7c018cec8d";
      var gemRepository = new GemHeaderRepository();
      var bandwithMapRepository
          = new BandwithMapRepository();

      var bwMap = new BandwithMap
      {
        GtcId = gtcId,
        // Obsah bw mapy
      };

      await bandwithMapRepository
          .InsertOneAsync(bwMap);

      var gem = new GemHeader
      {
        GtcId = gtcId,
        // Obsah GEM
      };

      await gemRepository.InsertOneAsync(gem);
    }
  }
}
```

7 Optimalizace a měření rychlosti

Největším požadavkem kladeným na databázi je požadavek na rychlost jejího zápisu. Za tímto účelem bylo provedeno několik měření s použitím obou programovacích jazyků a různých testovacích scénářů. Cílem výkonnostního testování je určit, který z obou použitých jazyků dosahuje lepších výsledků a jakým způsobem použít knihovny popsané v kapitole 6, aby bylo dosaženo co největšího výkonu. Jak již bylo popsáno v kapitole 5, minimální požadovaná rychlost je 76 000 GEM dokumentů za sekundu, avšak dá se očekávat zátěž kolem 152 000 GEM dokumentů.

7.1 Složení dat

K dosažení co nejrealističtější simulace reálného provozu byl u každého scénáře prováděn zápis všech tří typů dokumentů tj. GTC, GEM i bandwidth mapy. Mezi dokumenty reprezentující GTC a bandwitdh mapou existuje vazba 1:1 a mezi GTC a GEM dokumenty vazba 1:N. Lze tedy očekávat, že GEM dokumentů bude několikanásobně více než ostatních, a rychlost zápisu bude limitována právě zápisem GEM dokumentů.

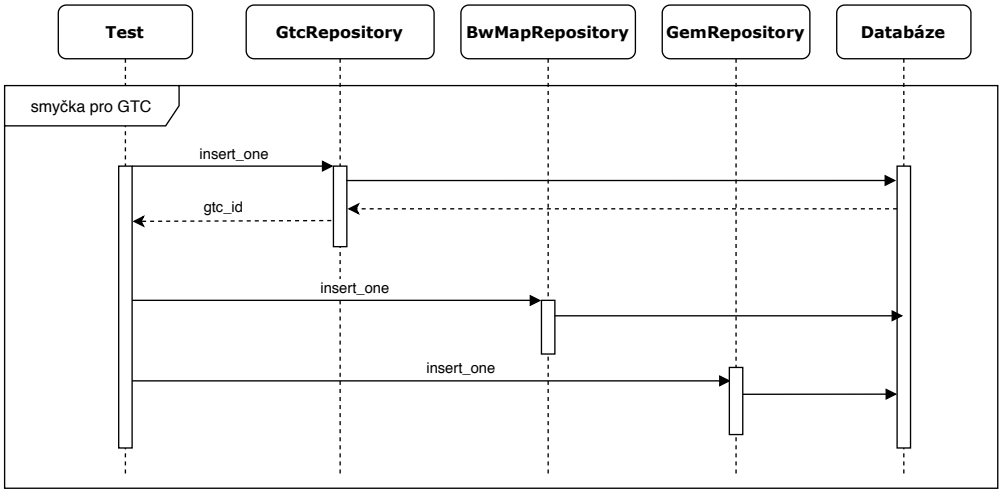
Ve všech testovacích scénářích je použit konstantní počet GTC hlaviček 1000 a položek v bandwidth mapě 10. Počet GEM hlaviček v jednom GTC rámci se mění od 10 do 800, aby bylo zjištěno, jak se rychlost zápisu vyvíjí v závislosti na tomto klíčovém parametru.

Testovací data jsou pro všechny scénáře i oba programovací jazyky stejná. V Pythonu se o generování data stará skript *data_generators.py* uvedený v příloze D a v jazyce C# se o generování starají třídy *GtcHeaderGenarator*, *GemHeaderGenarator* a *BandwithMapGenerator* uvedené v příloze E.

7.2 Testované scénáře

7.2.1 Sériový zápis

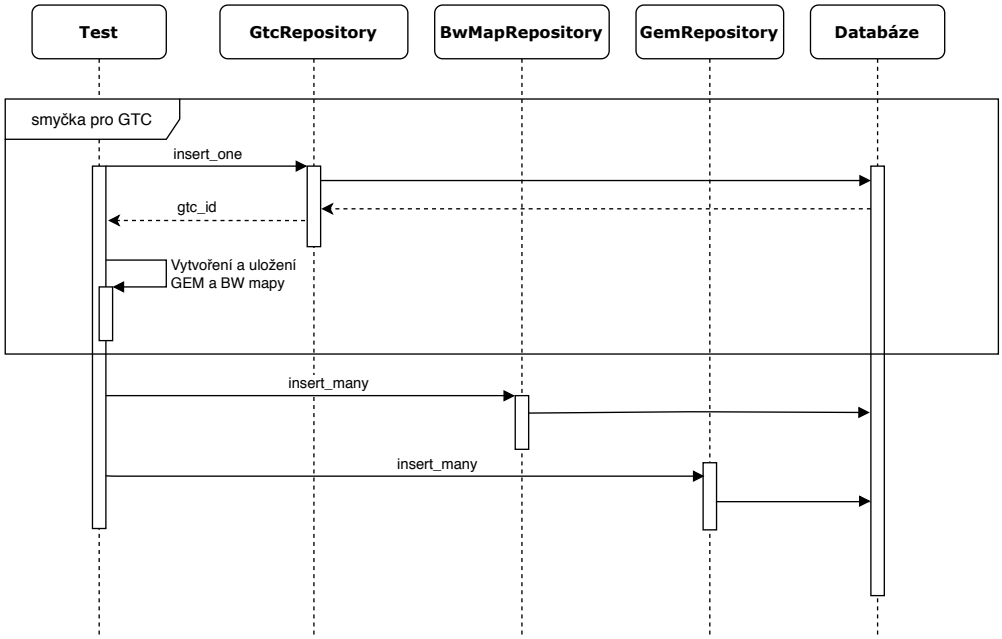
Nejjednoduším testovaným scénářem je sériový zápis všech dokumentů jeden po druhém. To znamená, že nejdříve je zapsán jeden GTC dokument, poté jedna bandwidth mapa a poté jeden GEM. Celá situace je znázorněna na sekvenčním diagramu 7.1.



Obr. 7.1: Výkonnostní testování - Sekvenční zápis.

7.2.2 Hromadný zápis všech GEM a BW map

Další testovaný scénář již využívá hromadný zápis. GTC hlavičky jsou i nadéle ukládány po jedné, neboť GEM i BW mapa vyžadují identifikátor GTC dokumentu vygenerovaný databází. Avšak BW mapa i GEM se neuloží hned po GTC, namísto toho se vloží do polí, která jsou hromadně uložena až po všech GTC objektech. Dochází tak pouze k jednomu volání *BwMapRepository* a *GemRepository*. Scénář je znázorněn na diagramu 7.2.



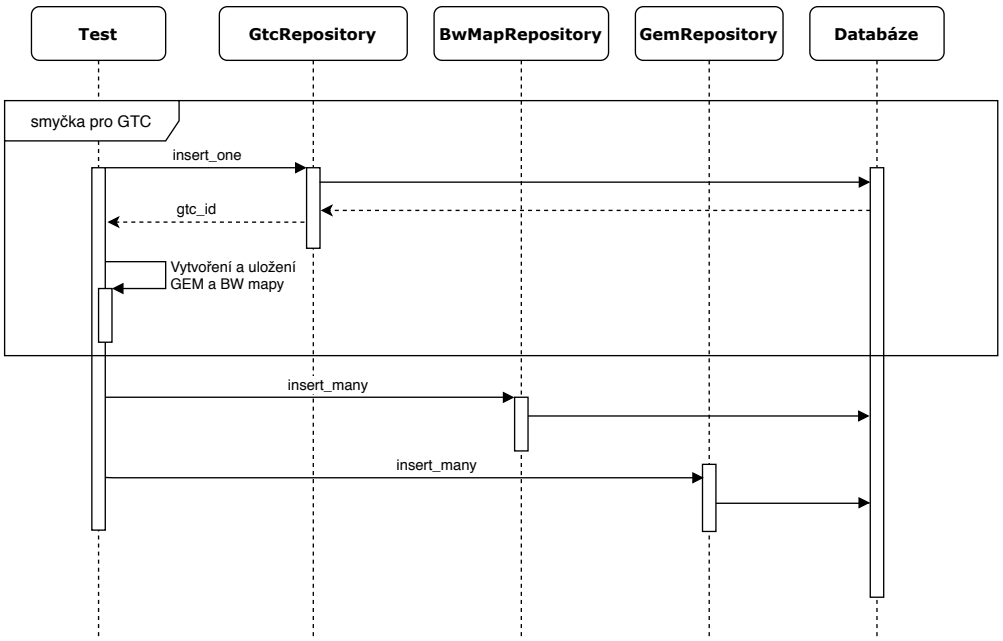
Obr. 7.2: Výkonnostní testování - Hromadný zápis.

7.2.3 Hromadný zápis po GTC blocích

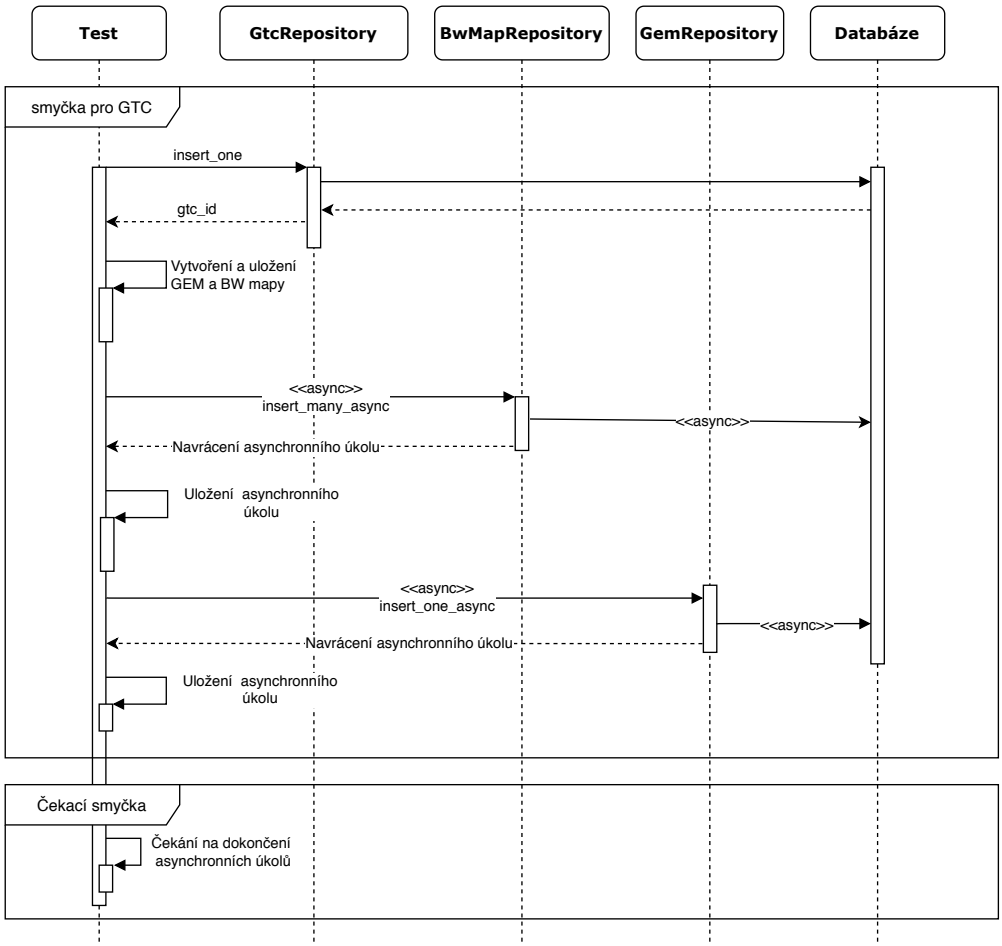
Tento scénář je velmi podobný předchozímu. Liší se tím, že zapisovány nejsou všechny BW mapy a GEM dokumenty najednou, nýbrž po skupinách náležících do daného GTC rámce. Scénář je znázorněn na diagramu 7.3.

7.2.4 Asynchronní hromadný zápis po GTC blocích

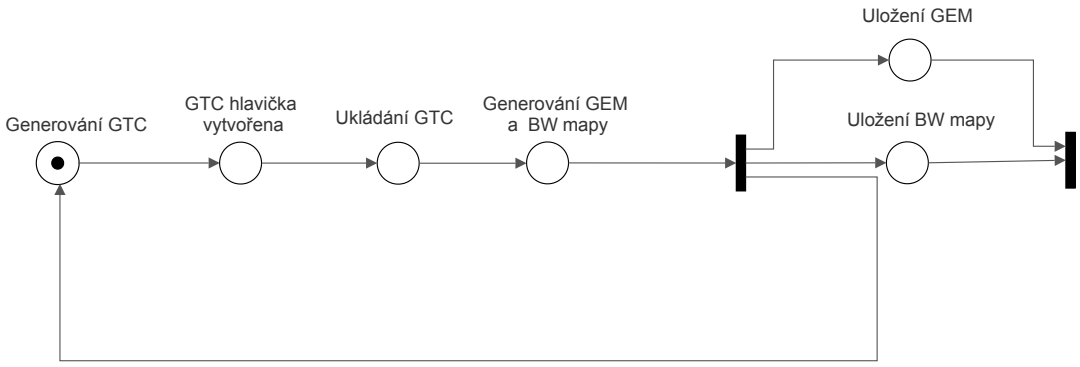
V tomto scénáři se využívá hromadný zápis po GTC blocích stejně jako ve scénáři 7.2.3. Rozdíl spočívá v použití asynchronních repozitářů. Namísto toho, aby výkonostní test čekal na uložení dat, dojde k uložení asynchronního úkolu, na který se čeká až na konci celého testu. Dochází tak k paralelizaci úkolů pro různé GTC rámce. Operace uvnitř jednoho GTC rámce však stále zůstávají sériové. Scénář je znázorněn na sekvenčním diagramu 7.4 a Petriho síti 7.5.



Obr. 7.3: Výkonnostní testování - Hromadný zápis po GTC blocích.



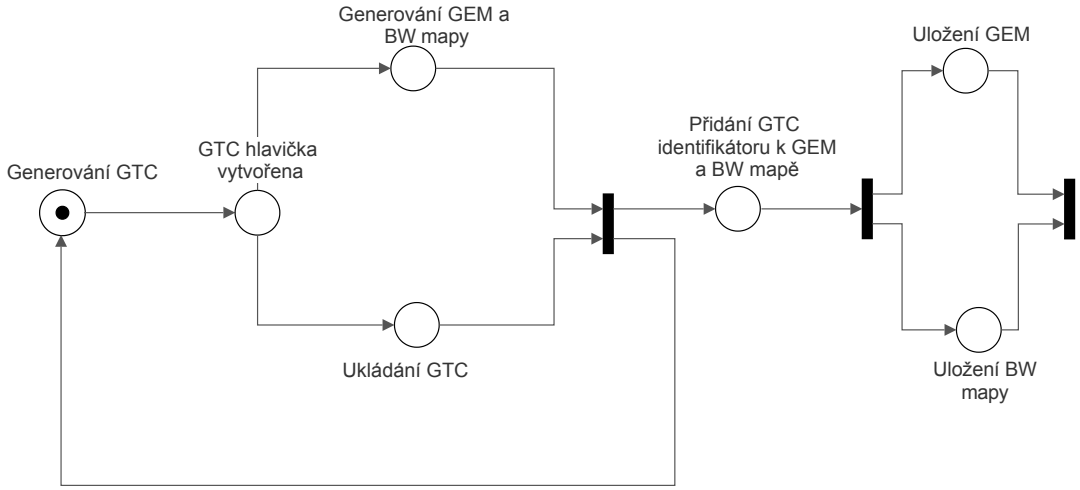
Obr. 7.4: Výkonnostní testování - Sekvenční diagram pro asynchronní hromadný zápis po GTC blocích.



Obr. 7.5: Výkonnostní testování - Petriho síť pro asynchronní hromadný zápis po GTC blocích.

7.2.5 Asynchronní hromadný zápis s paralelním generováním

Tento scénář rozšiřuje předchozí 7.2.4 a jde v paralelizaci ještě o krok dále. Paralelizuje operace i v rámci jednoho GTC rámce, a to tak, že paralelně běží ukládání GTC i generování GEM a bandwidth mapy. V okamžiku, kdy dojde k dokončení obou těchto operací, se do GEM a BW map přidá GTC identifikátor. Scénář je znázorněn na Petriho sítí 7.6.



Obr. 7.6: Výkonnostní testování - Asynchronní hromadný zápis s paralelním generováním.

7.3 Použitý hardware

Hardware použitý k výkonnostnímu testování je uveden v tabulce 7.1. Díky vysokému počtu fyzických jader procesoru se jedná o počítač vhodný k testování paralelismu a asynchronního zpracování.

Tab. 7.1: Parametry hardwaru použitého pro testování

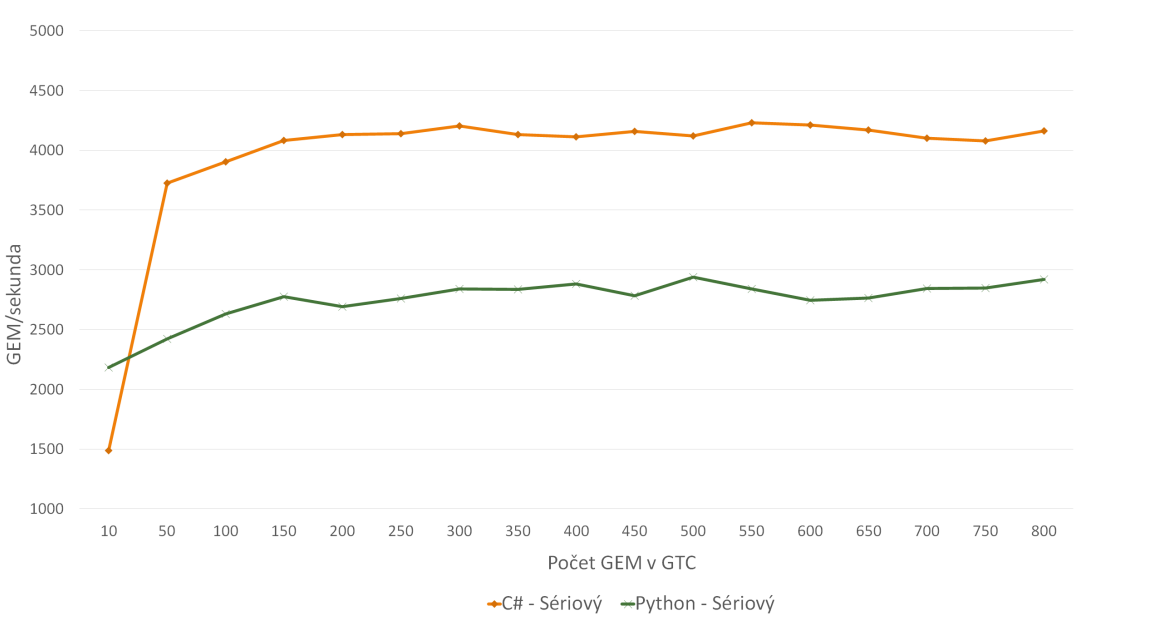
Komponenta	**Typ**	**Klíčové parametry**
Procesor	AMD Ryzen 7 2700X	3,2 GHz, 8 fyzických jader
RAM	HyperX Predator 4x8Gb	32 GB DDR4 3333 MHz
Disk	M.2 SSD WD SN500	Zápis 1450 MB/s

7.4 Výsledky měření

Tato kapitola obsahuje souhrnné výsledky z výkonnonstního měření. Kompletní výsledky měření jsou k nalezení v příloze F.

7.4.1 Sériový zápis

Rychlost sériového zápisu se u jazyka Python pohybovala přibližně v rozmezí 2100 až 2900 GEM dokumentů za sekundu. U jazyka C# bylo ve většině případů dosaženo vyšší rychlosti zápisu pohybující se v rozsahu 1500 až 4100 dokumentů za sekundu. Závislost rychlosti zápisu na počtu GEM dokumentů je vidět v grafu 7.7. Sériový zápis se nejeví jako vhodný, protože jeho rychlost je mnohonásobně menší než minimální požadovaná.

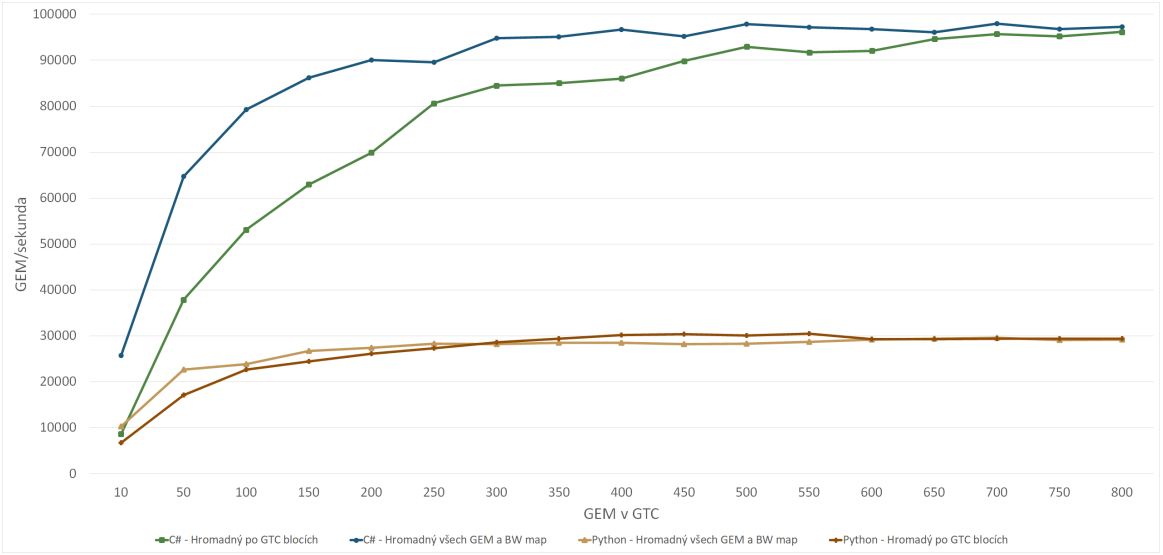


Obr. 7.7: Měření rychlosti sériového zápisu

7.4.2 Hromadný zápis

Rychlost hromadného zápisu se u Pythonu pohybovala v rozmezí 6000 až 32000 GEM hlaviček za sekundu a u jazyka C# přibližně od 8500 do 97000 GEM hlaviček za sekundu. Jak je patrné z grafu 7.8, hromadný všech zápis všech GEM dokumentů popsaný v scénáři 7.2.2 je pro menší počty GEM rychlejší. Při počtu nad 250 GEM se výkonnostní rozdíl začíná zmenšovat a pro 800 prvků je takřka zanedbatelný.

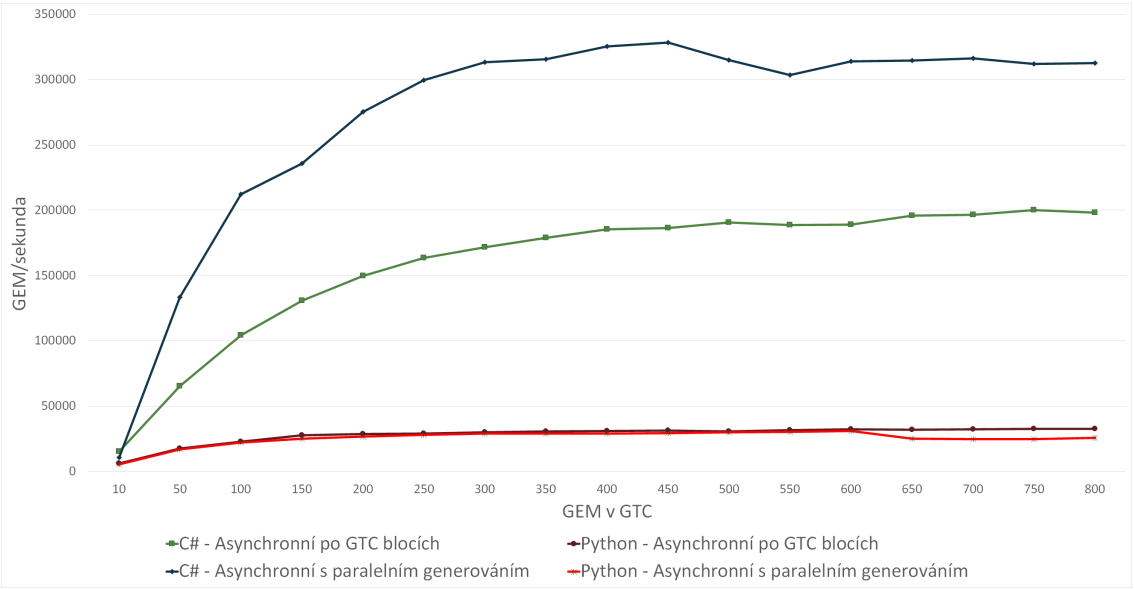
Rychlost v jazyce Python je i u těchto scénářů pod minimální požadovanou rychlostí. U jazyka C# již bylo pro vysoké počty GEM dosaženo minimální požadované hranice, ovšem očekáváné rychlosti 152 000 GEM za sekundu nebylo dosaženo.



Obr. 7.8: Měření rychlosti hromadného zápisu

7.4.3 Asynchronní zápis

Rychlost hromadného zápisu se u Pythonu pohybovala v rozmezí 6000 až 32000 GEM hlaviček za sekundu a u jazyka C# přibližně od 10000 do 325000 GEM hlaviček za sekundu. Asynchronní ukládání v jazyce Python nepřineslo téměř žádné zrychlení oproti hromadnému zápisu. Výkonnostní testování v jazyce C# ukazuje, že asynchronní přístup není vhodný pro malé počty GEM hlaviček do 50, kdy je rychlost zápisu nižší než u hromadného zápisu. Avšak pro větší množství dochází k výraznému nárůstu výkonu, kdy je u obou scénářů dosaženo minimální i očekávané rychlosti zápisu. Jako zvláště nadějný se jeví poslední testovací scénář 7.2.5 využívající maximální paralizaci.



Obr. 7.9: Měření rychlosti asynchronního zápisu

8 Závěr

Účelem této práce je seznámení se s optickými sítěmi GPON a NG-PON, databázovým systémem MongoDB a následná implementace databázového schématu a sady Python skriptů určených pro uložení komunikace na GPON sítch.

MongoDB je nerelační databáze založená na vysoké dostupnosti, škálovatelnosti a flexibilitě schématu pracující s formátem JSON. Schéma databáze bylo založeno především na základě požadavku rychlého ukládání dat, které je pro optické sítě zcela nezbytné, a již existujícího formátu, se kterým pracuje aplikace monitorující komunikaci na GPON sítích.

Pro navrženou databázi byla vytvořena sada Python skriptů a C# knihoven využívající synchronní i asynchronní přístup k databázi.

Minimální nezbytná rychlost zápisu, kterou aplikace musí zvládat, je 76 000 GEM dokumentů za sekundu. Lze však očekávat i zátěž dvakrát větší. Za účelem nalezení nejoptimálnějšího způsobu přístupu k databázi bylo v obou jazycích testováno pět scénářů. Nejvyšší dosažená rychlost sériového zápisu se v jazyce Python pohybovala kolem 2 900 GEM dokumentů za sekundu a v jazyce C# 4 100 GEM dokumentů za sekundu. U hromadného zápisu došlo k zrychlení až na 32 000 dokumentů za sekundu v Pythonu a 97 000 dokumentů za sekundu v jazyce C#. Další testovací oblastí byla rychlost asynchronního zápisu. Asynchronní implementace v jazyce Python nevykázala zrychlení oproti hromadnému zápisu. Oproti tomu v jazyce C# došlo k výraznému nárůstu rychlosti zápisu. Implementace v jazyce C# kombinující asynchronní hromadný zápis spolu s paralelním generováním databázových dokumentů dosáhla rychlosti až 325 000 GEM dokumentů za sekundu.

Výsledkem práce je databázové schéma a sada knihoven v jazyce Python a C# pro uchování komunikace z GPON sítí. Výkonnostní testování prokázalo, že řešení v jazyce C# je v vhodné i pro reálný provoz.

Literatura

[1] HOOD D., TROJER E. Gigabit-capable passive optical networks. Hoboken: Wiley, 2011

[2] HOLUBOVÁ I., KOSEK J., MINAŘÍK K., NOVÁK D. Big Data a NoSQL databáze. Praha: Grada, 2015. Profesionál. ISBN 978-80-247-5466-6.

[3] LEAVITT, N. Will NoSQL Databases Live Up to Their Promise? [online], 2010 [cit. 2019-12-14] Dostupné z: <http://www.leavcom.com/pdf/NoSQL.pdf>

[4] CONOLLY T, BEGG C, HOLOWCZAK R. Mistrovství - Databáze. Profesionání průvodce tvorbou efektivních databází, Praha, 2009

[5] Optické přístupové sítě [online]. Brno: e-Publi [cit. 2017-11-29]. Dostupné z: https://publi.cz/books/185/11.html

[6] ITU-T G.984.1 Gigabit-capable passive optical networks (G-PON): General characteristics.[online]. Ženeva, Švýcarsko, 2008 [cit. 2019-12-14]. Dostupné z URL: <https://www.itu.int/rec/T-REC-G.984.1-200803-I/en>.

[7] ITU-T G.984.2 Gigabit-capable passive optical networks (G-PON): Physical Media Dependent (PMD) layer specification.[online]. Ženeva, Švýcarsko, 2003 [cit. 2019-12-14]. Dostupné z URL: <https://www.itu.int/rec/T-REC-G.984.1-200803-I/en>.

[8] ITU-T G.984.3 Gigabit-capable passive optical networks (G-PON): Transmission convergence layer specification.[online]. Ženeva, Švýcarsko, 2014 [cit. 2019-12-14]. Dostupné z URL: <https://www.itu.int/rec/T-REC-G.987.1-201001-I>.

[9] BANKER K., BAKKUM K., VERCH S., GARRETT D., HAWKINS T. MongoDB in Action [online]. Shelter Island, USA, 2012. [cit. 2019-12-14] Dostupné z: <http://img105.job1001.com/upload/adminnew/2015-04-07/1428394945-PHQK1Q5.pdf>

[10] Redis Documentation [online]. Redis [cit. 2020-05-23] Dostupné z <https://redis.io/documentation>

[11] Couchbase Documentation [online]. Couchbase NoEQUAL [cit. 2020-05-23] Dostupné z <https://docs.couchbase.com/home/index.html>

[12] Amazon DynamoDB Developer Guide [online]. Amazon Web Services [cit. 2020-05-23] Dostupné z <https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Introduction.html

[13] Apache Cassandra Architecture [online]. Apache Software Foundation [cit. 2020-05-23] Dostupné z <https://cassandra.apache.org/doc/latest/architecture/index.html>

[14] Apache HBase ™ Reference Guide [online]. Apache Software Foundation [cit. 2020-05-23] Dostupné z <https://hbase.apache.org/book.html>

[15] OBJECTIVITY/DB TECHNICAL DETAILS & SUPPORT [online]. Objectivity Inc. [cit. 2020-05-23] Dostupné z <https://www.objectivity.com/products/objectivitydb/objectivitydbdetails/>

[16] ZODB - a native object database for Python [online]. Zope Foundation [cit. 2020-05-23] Dostupné z <http://www.zodb.org/en/latest/>

[17] The Neo4j Operations Manual v4.0 [online]. Neo4j [cit. 2020-05-23] Dostupné z <https://neo4j.com/docs/operations-manual/current/>

[18] HORVATH T., JURCIK M. Vaclav OUJEZDSKY V., ŠKORPIL V. GPON Analyzer - Frame Parser Module. [cit. 2020-05-27] Dostupné z URL:<https://ieeexplore.ieee.org/document/8768882>

[19] MongoDB Documentation [online]. MongoDB Inc. [cit. 2019-14-12]. Dostupné z: <https://docs.mongodb.com/>.

[20] Passive optical network [online]. San Francisco: Wikipedia, 2019 [cit. 2019-14-12]. Dostupné z: <https://en.wikipedia.org/wiki/Passive_optical_network>.

[21] GPON Network Components - Features and Advantages [online]. Longwood 2019 [cit. 2019-14-12]. Dostupné z: <https://www.multicominc.com/gpon-network-components-features-and-advantages/>.

[22] IETF RFC-8259 The JavaScript Object Notation (JSON) Data Interchange Format. [online]. [cit. 2019-14-12] Dostupné z: <https://tools.ietf.org/html/rfc8259>

[23] Python 3.8.3 documentation [online]. Dostupné z: <https://docs.python.org/3/>

[24] NASH T., C# 2010 Rychlý průvodce novinkami a nejlepšími postupy, Brno 2010

[25] C# Programming guide [online]. Microsoft [cit. 2020-05-26] Dostupné z: <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/>

[26] PyMongo 3.10.1 Documentation [online]. [cit. 2020-05-26] Dostupné z: <https://pymongo.readthedocs.io/en/stable/>

[27] Motor: Asynchronous Python driver for MongoDB. [online]. [cit. 2020-05-26] Dostupné z: <https://motor.readthedocs.io/en/stable/>

[28] Popularity of programming languages. [online]. Microsoft [cit. 2020-05-26] Dostupné z: <http://pypl.github.io/>

[29] Python Documentation - Coroutines and Tasks. [online] [cit. 2020-05-26] Dostupné z: <https://docs.python.org/3/library/asyncio-task.html>

Seznam symbolů, veličin a zkratek

API Application Programming Interface
APON ATM PON
BPON Broadband PON
BSON Binary JSON
BWmap Bandwith map
CRC Cyclic redundancy check
CRUD Create, Read, Update, Delete
FEC Forward Error Correction
GEM GPON Encapsulation Method
GPON Gigabit Passive Optical Network
GTC GPON Transmission Convergence
HEC Hybrid Error Correction
IoT Internet of Things
JSON JavaScript Object Notation
NoSQL Non SQL
OLT Optical Line Termination
ONT Optical Network Termination
OMCI ONU Management and Control Interface
ONU Optical Network Unit
PON Passive Optical Network
TDM Time-division multiplex
SDU Service Data Unit
SQL Structured Query Language

Seznam příloh

A GTC Validátor

Výpis A.1: Validátor GTC kolekce.

```
{
  "$jsonSchema":
  {
    "bsonType": "object",
    "required": ["Psync", "Identification", "PLOAMdownstream", "BIP"
    "properties": {
    "PSync": { "bsonType": "int"},
    "Identification": {
      "bsonType": "object",
      "properties": {
        "FEC": {"bsonType": "bool"},
        "Reserved": {"bsonType": "bool"},
        "SuperframeCounter": {"bsonType": "int"}
      }
    },
  "BIP": { "bsonType": "int"},
  "Plend": {"bsonType": "array"}
  }
}
```

B Synchronní Python repozitáře

Výpis B.1: Synchronní Python repozitáře

```
import pymongo
from bson.objectid import ObjectId
from typing import List
from pymongo.collection import Collection
from DataAccess.gpon_database_settings import GponDatabaseSettings


class MongoRepository:
  def __init__(self, collection: pymongo.collection):
    self.collection: pymongo.collection = collection

  def insert_many(self, items):
    self.collection.insert_many(items)

  def insert_one(self, item):
    return self.collection.insert_one(item).inserted_id

  def update_one(self, item):
    self.collection.replace_one({"_id": item["_id"]}, item)

  def update_many(self, items):
    for item in items:
      self.update_one(item)

  def delete_by_id(self, id: ObjectId):
    self.collection.delete_one({"_id": id})

  def delete_by_ids(self, ids: List[ObjectId]):
    self.collection.delete_many({"_id": {"$in": ids}})

  def get_by_id(self, id: ObjectId):
    return self.collection.find_one({"_id": id})

  def find(self, item_filter):
    return self.collection.find(item_filter)


class GtcRepository(MongoRepository):

  def __init__(self, client: pymongo.MongoClient):
    collection: Collection = client.get_database(GponDatabaseSettings.database_name)\
      .get_collection(GponDatabaseSettings.gtc_collection)
    super().__init__(collection)


class GemRepository(MongoRepository):

  def __init__(self, client: pymongo.MongoClient):
    collection: Collection = client.get_database(GponDatabaseSettings.database_name)\
      .get_collection(GponDatabaseSettings.gem_collection)
    super().__init__(collection)

  def get_by_gtc_id(self, gtc_id: ObjectId):
    return self.find({"GtcId": gtc_id})


class BwMapRepository(MongoRepository):

  def __init__(self, client: pymongo.MongoClient):
    collection: Collection = client.get_database(GponDatabaseSettings.database_name)\
      .get_collection(GponDatabaseSettings.bwmap_collection)
    super().__init__(collection)

  def get_by_gtc_id(self, gtc_id: ObjectId):
    result = self.find({"GtcId": gtc_id})
    if result.count() > 0:
      return result[0]

    return None
```

C Asynchronní Python repozitáře

Výpis C.1: Asynchronní rodičovský Python repozitář

```
import pymongo
from bson.objectid import ObjectId
from typing import List
import motor.motor_asyncio
import asyncio
from DataAccess.gpon_database_settings import GponDatabaseSettings


class MongoRepositoryAsync:
  def __init__(self, collection: pymongo.collection):
    self.collection: motor.motor_asyncio.AsyncIOMotorCollection = collection

  async def insert_many_async(self, items):
    return self.collection.insert_many(items)

  async def insert_one_async(self, item):
    return await self.collection.insert_one(item)

  async def update_one_async(self, item):
    await self.collection.replace_one({"_id": item["_id"]}, item)

  async def update_many_async(self, items):
    tasks = []
    for item in items:
      tasks.append(self.update_one_async(item))

    await asyncio.wait(tasks)

  async def delete_by_id_async(self, id: ObjectId):
    await self.collection.delete_one({"_id": id})

  async def delete_by_ids_async(self, ids: List[ObjectId]):
    await self.collection.delete_many({"_id": {"$in": ids}})

  async def get_by_id_async(self, id: ObjectId):
    return await self.collection.find_one({"_id": id})

  def find_async(self, item_filter) -> motor.motor_asyncio.AsyncIOMotorCursor:
    # This function returns asynchronous cursor
    # Usage:
    # cursor = repository.find({})
    # await cursor.to_list(max_items)

    return self.collection.find(item_filter)
```

Výpis C.2: Asynchronní dceřinné Python repozitáře

```
class GtcRepositoryAsync(MongoRepositoryAsync):

  def __init__(self, client: motor.motor_asyncio.AsyncIOMotorClient):
    collection = client.get_database(GponDatabaseSettings.database_name) \
      .get_collection(GponDatabaseSettings.gtc_collection)

    super().__init__(collection)


class GemRepositoryAsync(MongoRepositoryAsync):

  def __init__(self, client: motor.motor_asyncio.AsyncIOMotorClient):
    collection = client.get_database(GponDatabaseSettings.database_name)\
      .get_collection(GponDatabaseSettings.gem_collection)

    super().__init__(collection)

def get_by_gtc_id_async(self, gtc_id: ObjectId) -> motor.motor_asyncio.AsyncIOMotorCursor:
  # This function returns asynchronous cursor
  # Usage:
  # cursor = repository.find({})
  # await cursor.to_list(max_items)
  return self.find_async({"GtcId": gtc_id})


class BwMapRepositoryAsync(MongoRepositoryAsync):

  def __init__(self, client: motor.motor_asyncio.AsyncIOMotorClient):
    collection = client.get_database(GponDatabaseSettings.database_name) \
      .get_collection(GponDatabaseSettings.bwmap_collection)

    super().__init__(collection)

  async def get_by_gtc_id_async(self, gtc_id: ObjectId):
    result = await self.find({"GtcId": gtc_id}).to_list(1000)
    if result.count() > 0:
      return result[0]

    return None
```

D Generátor testovacích dat v jazyce Python

Výpis D.1: Asynchronní Python repozitáře

```
    def createGtcHeader():
      return {
        "Psync": 123,
        "Identification": {
          "FEC": True,
          "Reserved": False,
          "SuperframeCounter": 994141217
        },
        "PLOAMdownstream": {
          "ONUid␣": 255,
          "MessageID␣": 11,
          "Data": "",
          "CRC": 158
        },
        "BIP": 196,
        "Plend": [{
          "Blen": 6,
          "Alen": 0,
          "CRC": 245
        }, {
          "Blen": 6,
          "Alen": 0,
          "CRC": 245
        }]
      }

    def createBandwithMap(gtcID):
      return {
        "GtcId": gtcID,
        "Bwmap": [{
            "AllocID": 1,
            "Flags": 1,
            "StartTime": 1,
            "StopTime": 1,
            "CRC": 1
          }, {
            "AllocID␣": 1,
            "Flags": 1,
            "StartTime": 1,
            "StopTime": 1,
            "CRC": 1
        }]
      }

    def createGemHeader(gtcID):
    return {
      "GtcId": gtcID,
      "PLI": 1,
      "PortId": 1,
      "HEC": 1,
      "PTI": {
        "LastFragment": False,
        "UserData": True
      }
    }
\end{minipage}}
```

E Generátor testovacích dat v jazyce C#

Výpis E.1: Generátor testovacích GTC hlaviček v jazycek C#

```
using UDP_parser;
namespace GPON.PerformanceTesting.Generators
{
  internal static class GtcHeaderGenerator
  {
    public static GPONFrame[] CreateMultiple(int count)
    {
      var frames = new GPONFrame[count];
      for (int i = 0; i < count; i++)
      {
        frames[i] = CreateOne();
      }

      return frames;
    }

    public static GPONFrame CreateOne()
    {
      return new GPONFrame
      {
        PCBdownstream = new GPONFrame.PCBd
        {
          Psync = 123,
          BIP = 196,
          Identification = new GPONFrame.PCBd.Ident
          {
            FEC = 1,
            Reserved = 1,
            SuperframeCounter = 994141217
          },
          Plend = new GPONFrame.PCBd.PLend[]
          {
            new GPONFrame.PCBd.PLend
            {
              Blen = 6,
              Alen = 0,
              CRC = 245
            },
            new GPONFrame.PCBd.PLend
            {
              Blen = 6,
              Alen = 0,
              CRC = 245
            },
          }
        }
      };
    }
  }
}
```

Výpis E.2: Generátor testovacích GEM hlaviček v jazyce C#

```
using GPON.MongoDB.Entities;

namespace GPON.PerformanceTesting.Generators
{
  public static class GemHeaderGenerator
  {
    public static GemHeader CreateOne(string gtcId)
    {
      return new GemHeader
      {
        GtcId = gtcId,
        PortId = 1,
        HEC = 1,
        PTI = new GemPti
        {
          LastFragment = false,
          UserData = true
        }
      };
    }

    public static GemHeader[] CreateMultiple(int count, string gtcId)
    {
      var frames = new GemHeader[count];
      for (int i = 0; i < count; i++)
      {
        frames[i] = CreateOne(gtcId);
      }

      return frames;
    }
  }
}
```

Výpis E.3: Generátor testovacích bandwidth map v jazyce C#

```
using GPON.MongoDB.Entities;
using static UDP_parser.GPONFrame.PCBd;

namespace GPON.PerformanceTesting.Generators
{
  internal static class BandwithMapGenerator
  {
    public static BandwithMap Generate(string gtcId, int mapCount)
    {
      var bwMaps = new BWmap[mapCount];
      for(int i = 0; i < mapCount; i++)
      {
        bwMaps[i] = new BWmap
        {
          AllocID = 1,
          Flags = 1,
          StartTime = 1,
          StopTime = 1
        };
      }

      return new BandwithMap
      {
        GtcId = gtcId,
        Maps = bwMaps
      };
    }
  }
}
```

F Kompletní data z výkonnostního měření

GEM v GTC	10	50	100	150	200	250	300	350	400	450	500	550	600	650	700	750	800
C# - Sériový	1488	3726	3903	4083	4131	4138	4206	4133	4114	4160	4121	4232	4211	4169	4102	4080	4163
C# - Hromadný všech	25707	64683	79302	86157	90050	89509	94817	95135	96642	95218	97867	97207	96790	96054	97943	96737	97312
C# - Asynchr. po GTC	15456	65274	104167	130776	149925	163399	171821	178937	185529	186258	190549	188615	189095	195783	196574	200160	198265
C# - Hromadný po GTC	8681	37908	53135	62946	69881	80619	84507	85034	86003	89838	92920	91697	92061	94614	95720	95214	96165
C# - Asynchr. s paralelním gen.	10905	133333	212314	235849	275482	299401	313152	315600	325468	328467	314861	303532	313972	314618	316170	311850	312744
Python - Sériový	2182	2422	2632	2776	2693	2760	2842	2835	2883	2784	2940	2840	2747	2764	2844	2849	2919
Python - Hromadný všech	10300	22705	23856	26738	27466	28311	28198	28527	28557	28232	28341	28708	29254	29418	29617	29129	29168
Python - Hromadý po GTC	6751	17108	22655	24487	26188	27339	28655	29434	30161	30398	30075	30494	29299	29292	29435	29454	29439
Python - Asynchr. po GTC	6278	17652	22915	27830	28893	29111	29943	30674	31052	31396	30612	31608	32316	32149	32466	32760	32784
Python - Asynchr. s paralelním gen.	5520	16991	22301	25187	26893	28001	29071	29116	29022	29520	30257	30494	31028	25303	24946	24911	25959